Dynamics in large scale networks

John Clements

Mathematics & Statistics

Submitted in partial fulfillment

of the requirements for the degree of

M.Sc.

Faculty of Mathematics and Science, Brock University

St. Catharines, Ontario

Abstract

In this thesis we study the properties of two large dynamic networks, the competition network of advertisers on the Google and Bing search engines and the dynamic network of friend relationships among avatars in the massively multiplayer online game (MMOG) Planetside 2. We are particularly interested in removal patterns in these networks. Our main finding is that in both of these networks the nodes which are most commonly removed are minor near isolated nodes. We also investigate the process of merging of two large networks using data captured during the merger of servers of Planetside 2. We found that the original network structures do not really merge but rather they get gradually replaced by newcomers not associated with the original structures. In the final part of the thesis we investigate the concept of motifs in the Barabási-Albert random graph. We establish some bounds on the number of motifs in this graph.

Key words: large social network analysis, dynamic network analysis, graph motif, vertex/node removal networks, Motifs in the Barabási–Albert Algorithm.

Acknowledgements

I would like to thank my supervisors Dr. Babak Farzad and Dr. Henryk Fukś for their guidance and support. I would also like to thank the developers and players of PlanetSide 2.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the last 20 years there has been a great deal of scientific interest in large scale networks [19]. A great deal of this research has focused on how these networks evolve over time. Many studies have sought to understand the local dynamics of these large structures through simple rules and models of the local structure of these large network [11, 13, 31].

Scientists have studied a large array of different real world networks to come to understand how things are connected at on the large scale [18, 27, 28]. As the internet matures and becomes increasingly integrated into the business and daily lives of people all over the world understanding large networks only grows more important.

Despite the great interest in these networks, the processes that affect

how nodes are removed have not been sufficiently examined. Models of large networks usually focus on understanding how nodes are added to the network and how links are formed and removed. This shortcoming is important because if we want a good model the way a network shrinks is as important as the way it grows, so we need to understand the entire life cycle of the nodes involved. This is especially significant if we wish to create applications around large networks since the design choices we make can impact how long nodes persist in the system.

The main portion of this thesis is concerned with the removal of nodes from large dynamic networks. We investigated two large real world datasets consisting of multiple networks in search of removal patterns. The competition networks of advertisers on the Google and Bing search engines and the dynamic network of friend relationships among avatars in the massively multiplayer online game (MMOG) Planetside 2.

Problems.

In order to find such a pattern we need to collect a sufficiently large database on our networks of nodes representing our actors and edges for the links between them. This database will need to be stored carefully in order to maintain its integrity.

Next, we must determine which actors have been removed (expired, died,

quit, etc.) or otherwise made irrelevant. This is one of the hardest problems we face, as finding exact definition of removed nodes will depend heavily on the network being examined. And finally we need to analyze our data by sifting through our networks for structural commonalities among the removed actors. Contrasting them with the structures of the other nodes is the final task which is required.

Layout of this thesis

The second chapter introduces both data sets in more detail, including a detailed description of how the information was collected and stored. Then we cover how we found the removed nodes in each data set. In what follows we explore the basic properties, such as the degree distribution, clustering coefficient and determine if the graph has the small world property.

Chapter three consists of a preliminary analysis of the dynamics of the PlanetSide 2 dataset, mostly covering the mixing patterns of the avatars over time. The correlations between both the various attributes and graph theoretic properties of the avatars are investigated as well.

Finally, Chapter four covers the results for node removal. We start with the basic properties that set the removed nodes apart from the rest of the network. The analysis also compares the importance of edge removal to that of node removal in the PlanetSide 2 data. Dynamic analysis of how

these patterns change over time is also included, along with discussion and conclusions.

In Chapter five, findings on the 8 month study of a server merger are presented. We explore the mixing process of the two servers through a series of snapshots graphically and analytically. Chapter six presents bounds on the possible network motifs for a given parameterization of any Barabási–Albert network.

Chapter seven summaries our conclusions and lists some potential future work. There are also two appendices. The first contains the source code used to gather the Planetside 2 data as well as source code for the more difficult portions of the analysis. The second appendix contains useful additional information and figures.

Definitions and terms

Unless otherwise stated these definitions are found in the textbook [6].

A *Graph* G is an ordered pair $(V(G), E(G))$ consisting of a set $V(G)$ of vertices and a set $E(G)$ of edges, that form connections between them.

The *degree* of a vertex v in a graph G, denoted $k_G(v)$ is the number of edges of G incident with v.

A u,v path is a simple graph whose vertices can be ranged in a linear sequence such that any two are adjacent if they are consecutive in the sequence, whose vertices of degree 1 (its endpoints) are u and v. The length of a path is the number of edges. The *distance* between two vertices u,v is the length of the shortest path between them.

The *diameter* of G is the greatest distance between two vertices of G.

For a unweighted graph the *clustering coefficient* of a node v is the number of links among nodes in the neighborhood of v divided by the number of possible links.

$$c_v = \frac{2T(v)}{k_v(k_v - 1)}$$

where $T(v)$ is the number of triangles (i.e. connected neighbors) v is involved in. The clustering coefficient of a degree 0 or 1 node is set as 0.

The average clustering coefficient of a graph is the average of the clustering coefficients of its vertices. [5]

Given n and p, generate graphs with n vertex by connecting each pair of vertices be a edge with probability p independently. Each graph with m edges has probability $p^m(1-p)^{\binom{n}{2}-m}$. Such graphs are called $G_{n,p}$ random graphs.

Chapter 2

Data

This chapter introduces our data sets. Starting with the network of competing businesses paying for ad space through the Google and Bing search engines. Followed by the network of friendships between avatars in the massively multiplayer online game (MMOG) Planetside 2. Each dataset has detailed explanations of how the data was gathered and structured, and the motivations behind looking into it. We discuss how we identify the removed nodes of failed businesses and abandoned avatars respectively. We also give a broad overview of their basic properties, including the number of nodes and edges, clustering coefficient, degree distribution, population trends and determine if the networks are small world.

Competition network

Our competition network consists of websites competing for ad space in Google and Bing search results in 2009. This network maps how companies were competing with each other, by connecting companies who were advertising in the same keyword search with weighted edges. The weights being equal to the number of different keyword searches they share. We then identified the websites that have since failed as of early 2014.

The motivation comes from economics, business can fail for a huge variety of reasons many of which involve pressures from their competitors. If two companies are competitors their products should appear in the same ad spaces, and once a business fails their website should end up being taken down or sold off. For example small companies competing directly with large corporations often have trouble staying in business. Or perhaps small companies in general have a higher rate of failure. If a large number of business competing in multiple areas then more should end up failing under the pressure. These kinds of events should be viable in our network of competition.

The Data

The original data for the competition network was gathered in 2009 by a former grad student. At the time Google and Bing searches had boxes for paid advertisements as seen in Figure 2.1.

The data was recorded in .txt files divided into Google results and Bing results. Each of these text file contains keyword searches related to a particular product or service like health care, legal advice or computer services. Each of the keyword searches was followed by the list 0 to 9[1] urls from the ad boxes; for example a Google search for "2 way baby monitor" returned the following urls:

- http://www.aartech.ca
- http://www.beso.com
- http://anextraeye.com
- http://www.shopzilla.com

This means that aartech, beso, anextraeye, and shopzilla are sharing ad space so we say they are competing and add edges between all of them in the network, or increase the weight by one if it already exists.

Once we have done this for all of our keyword searches we have to clean up the data by identifying all of the urls belonging to the same company. For example http://www.ebay.ca, https://www.ebay.com and, http://www.ebay/checkout/etc.ca are the same business so we combine all of those nodes together in the final graph and name it eBay. We ignore self loops when they occurred.

We also created a directed version where edges are directed from higher

[1]Google never returns more than 8, Bing never returns more than 9.

urls to lower in the ad box, if for example the first result is eBay is the second was craigslist we would set the edge to run from eBay to craigslist.

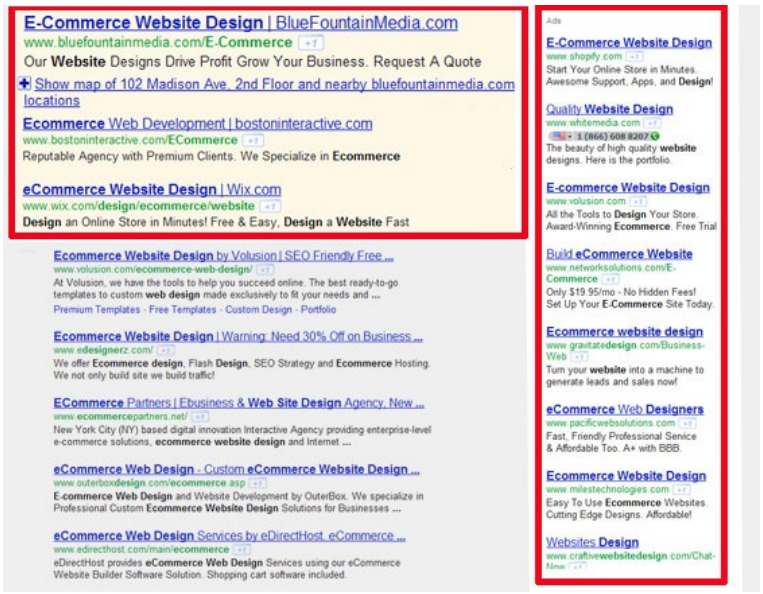


Figure 2.1: Example ad boxes.

Identifying Closed companies

Closed businesses were found by checking if the domain name was available for purchase. Because of anti-scripting measures this had to be done manually, we used the mass domain availability test provided by Godaddy to check 500 urls at a time. We verified this test against a few competing domain name availability websites on a sample of random urls and found no conflicts.

Competition network density and average clustering coefficient.

Since our competition networks consist of a single snapshot we summarize the average clustering coefficient and density plus the number of nodes and edges in Table 2.1.

	Bing	Google
Number of companies	10015	30510
Number of edges	160411	371958
Density	0.003199	0.000799
Average clustering coefficient	0.7687	0.7444

Table 2.1: Basic competition network information

The density of the Google network is less than a quarter that of the Bing network. While the average clustering coefficients are similar.

The average clustering coefficient is greater than the density in both of competition networks so they both demonstrate high clustering. This is partially because the network consists of cliques of companies, every result in a keyword search is connected to every other result in the same search. This means that any company that appears in only a single search will have a clustering coefficient of 1 because all of its competitors are mutually connected.

Small world properties in the competition networks

In the famous six degrees of separation experiment [16] they showed that a letter passed through a chain of acquaintances could sent to specific destination from considerable distances through a median of six steps. This is what is known as the small world phenomena. And social networks are commonly found to be small world.

Formally small world networks are defined as having small diameters, power law degree distributions and high clustering coefficients.

Diameter and average path length (APL).

In order to determine if the competition network is small world we will first look at the diameter and APL. By comparing the empirical values to the expected diameter of a $G_{n,p}$ random graph with the same number of nodes and edges we can determine if the diameter is small or not.

The diameter of a $G_{n,p}$ graph with a specific number of nodes n, and density p when $np > 1$ is known to be $D(G_{n,p}) \propto \frac{ln(n)}{ln(np)}$ [1].

The results for the diameter and average shortest path length of the empirical data and the expected values in a $G_{n,p}$ random graph are provided in Table 2.2.

Competition network			Random graphs.	
	Bing	Google		
Diameter	7	8	3	4
APL	2.528	2.752	2.945 ± 0.00180	5.108 ± 0.00696

Table 2.2: The diameter and average path length of the competition network

The competition networks have a short APL and a large diameter. Thus both the Bing and Google networks immediately fail to meet even the first criteria for a small world network.

Clustering coefficients

The following tables show the clustering coefficient for the two competition networks.

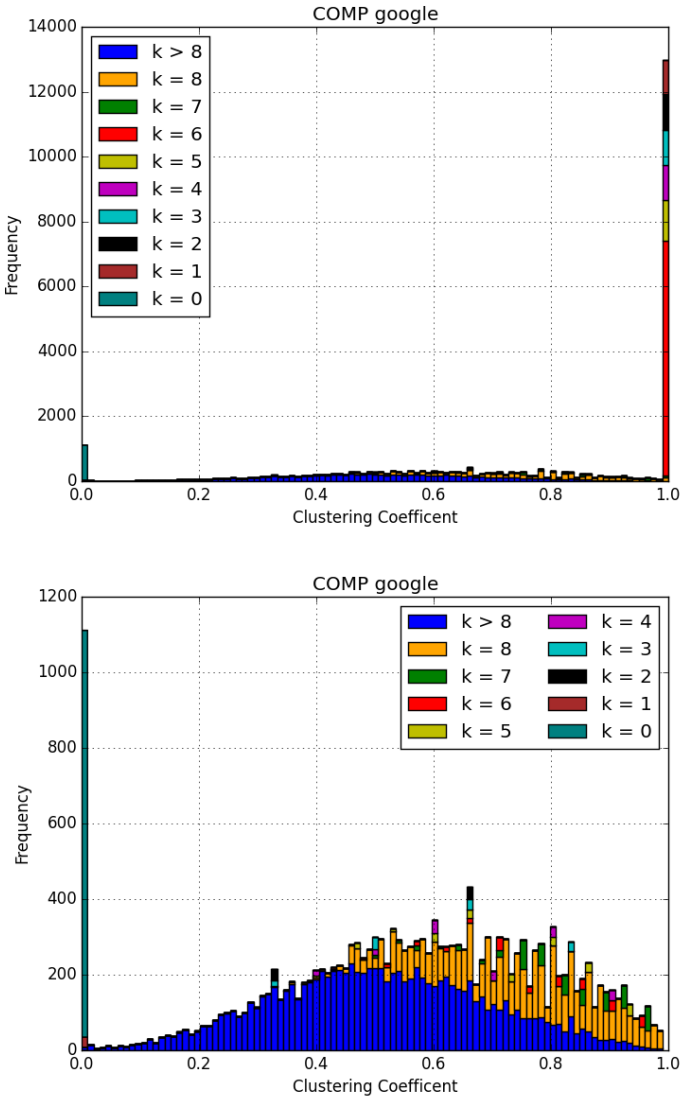


Figure 2.2: The clustering coefficient distribution of Google.

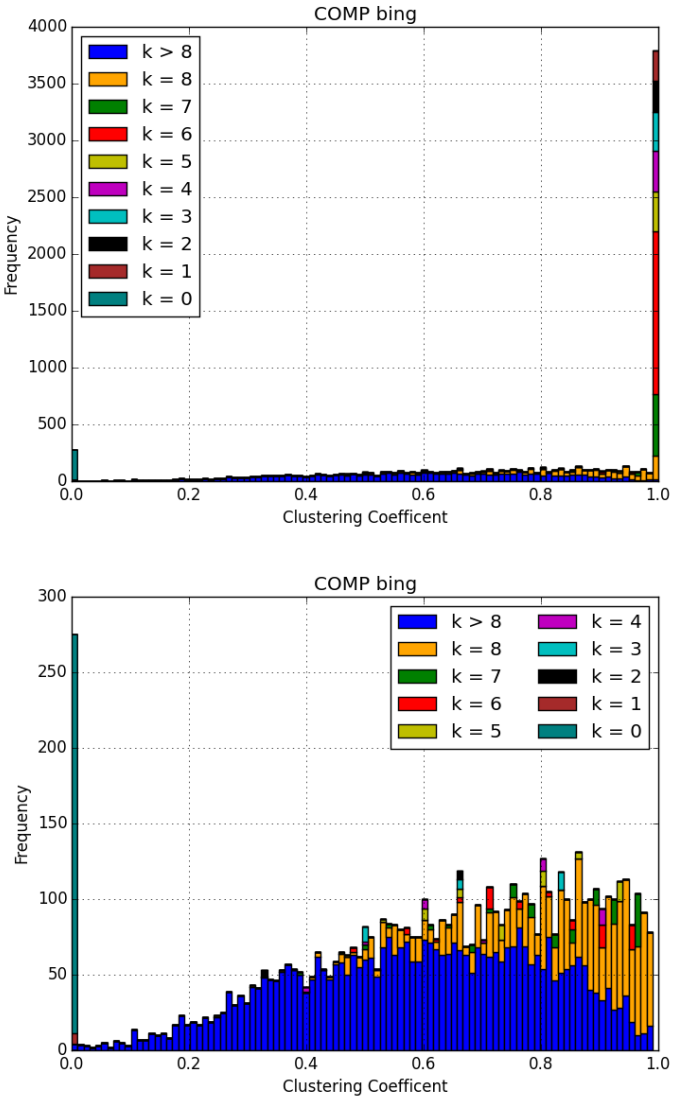


Figure 2.3: The clustering coefficient distribution of Bing.

As we can see in Figure 2.2 and Figure 2.3 the majority of companies have a clustering coefficient of one. This is because they only appeared in a single keyword search. In rest of the distribution we see that lower degree

companies are skewed towards higher clustering.

So while these companies are not isolated they are effectively peripheral, usually appearing in only a single keyword search. If we look at the rest of the distribution without the spike we can see that the competition networks tend to have higher clustering coefficients in general as a result of each company being part of a clique by default.

Competition network degree distribution

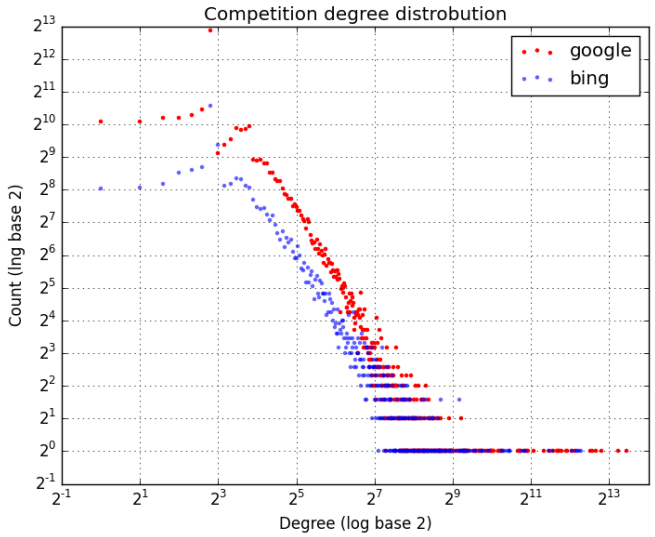


Figure 2.4: Competition network degree distribution

The degree distribution of the competition network is shown in Figure 2.4. The spikes around 8 and 16 consist of companies that appeared in one and two typical keyword searches respectively.

Confirming power law behavior

To show that a degree distribution is power law it is necessary but not sufficient that the data be linear in a log - log plot. Which as we can see in Figure2.4 is true but several alternate models have similar log - log plots [9]. So we used the methods in [2] to calculate whether the power law model is a better fit than the most likely alternatives.

We will compare the power law model $p(x) = x^{-\alpha}$ to the exponential $p(x) = e^{-\lambda x}$ and power law with exponential cutoff $p(x) = x^{-\alpha}e^{-\lambda x}$.

The quality of the fit uses the Kolmogorov Smirnov test (D) $D = \max_{x \geq xmin} |S(x) - P(x)|$ where $S(x)$ is the cumulative distribution function (CDF) of the data, and $P(x)$ is the CDF of the model [9].

The important step is identifying where the heavy tail starts by finding a value for xmin such that all $x >$ xmin are in the heavy tail, which is done by finding the value of xmin that minimizes D for the best fit for alpha for all x [9]. See Figure 2.12 for an example showing the minimization of D for the Google network.

Results

See Table 2.3 for the log likelihoods of the alternate models being superior to powerlaw. Both completion networks firmly reject an exponential fitting

Competition Networks				Log likelihood		
Network	xmin	D	α	$x^{-\alpha}$ vs $e^{-\lambda x}$ (R,p)	$x^{-\alpha}$ vs $x^{-\alpha}e^{-\lambda x}$ (R,p)	ntail
Bing	198	0.03752	2.3361	(44.618, 1.702e-4)	(-0.3621, 0.3948)	160
Google	200	0.02832	2.0609	(82.177, 8.448e-7)	(-1.6116, 0.0917)	230

Table 2.3: Fit and log likelihood for the two competition networks.

for the optimal xmin, and the p value is not small enough to say that a power law with exponential cutoff is a better fit then a power law. We conclude that the competition networks are heavy tailed and in that heavy tail they follow a power law.

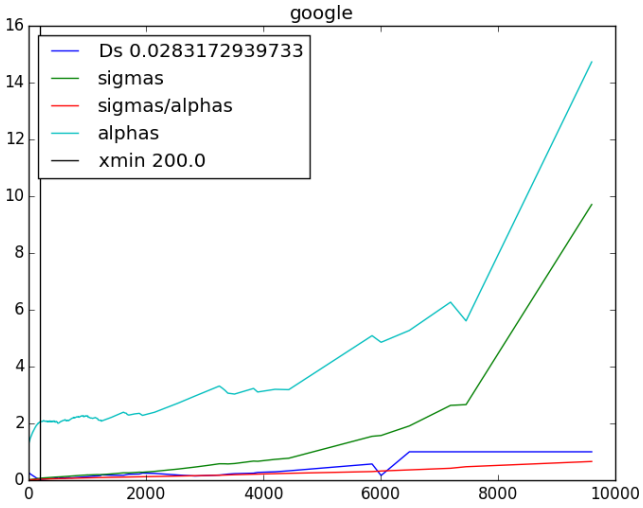


Figure 2.5: Google: Fits for all possible values of xmin.

Figure 2.5 shows the best fit for an $x^{-\alpha}$ power law distribution and the sigma error on alpha for all potential xmin for the Google data.

Conclusion

While the competition networks fulfill both the high clustering and power law degree distribution requirements. They do not have short diameters so they are not small world networks.

Planetside 2

Our Planetside 2 data takes the form of weekly [2] "snapshots" of the network of friendships between avatars in the computer game Planetside 2. Each snapshot is a graph with each avatar being a node and each edge representing a friend link. The edges are unweighted and undirected as friendships are mutual. Additional information such as names, faction, time played and avatar creation date were also collected for each avatar, to give context to the network and get a better picture of its dynamics.

The Data

We gathered all of the information from the Sony Online Entertainment (SOE) census API census using a our crawling algorithm. The API lets users query the SOE databases for information on several of the company's games, and has been used in earlier studies [23, 24]. The API is intended

[2] See Table 9.1 for exact dates

to support the creation of player made websites and tools for performance tracking statistics and other fan activities such as leader boards where avatars are ranked for example by performance or score. This API provides a great depth of information on individual avatars, however account information is not available for privacy reasons.

The SOE representative granted us permission under two conditions. They requested we not query any out of date avatars (avatars who have not been active for more than 44 days) and that we perform no more than 100 calls a minute.

So our crawl is limited to avatars online in the last 44 days, and because it is possible for avatars to add a friend on a different server (this is rare) we restrict the crawl to only friendships between avatars on the same server. The crawler typically takes two and a half hours to gather a snapshot for each single server.

A note on MMOG servers with a single exception (EVE Online) most MMOG run multiple parallel servers. Each is a self contained persistent copy of the game world environment with its own set of avatars. Usually servers will be spread across different geographic regions to reduce latency. We studied three servers the US west coast server Connery, the US east coast server Emerald and Miller one of the two EU servers. The remaining server is located in Australia and serves the oceanic region. In the case of Planetside 2 it is not possible for avatars to interact across servers except for private

messages, and friendship links, nor is it possible to transfer an avatar to a different server.

Snapshot algorithm

The code follows these steps:

1. The first step requires a list of active avatar Ids from the server we want to crawl. The version included in the index pulls avatars from the leader boards [3]. However in the original seed Ids were chosen manually. Once this is done create a queue of Ids to check and add the seed Ids to it.

2. Get the friend lists of all avatar Ids in the queue from the API. If the friendlist of a Id is successfully found remove it from the queue and add it to the list of visited avatar Ids.

3. Then go through each friend in those friend lists, sort out the valid Ids (activity in the last 44 days and on the correct server). Save each of the valid friend relationships to the edge set. If they are not in the list of checked Ids add each of these valid Ids to the queue.

4. If there are still Ids in the queue of Ids to check go to step two otherwise continue.

[3] The top 100 avatars for kill count and death count that week

5. Record the edge list as a sql table and terminate the API connection.

6. Get the list of all avatars found in the crawl and gather the avatar attributes for each of them.

The actual code is a bit more complex since we need to handle failed API calls, check data integrity and comply with the restrictions as well as record it into our sqlite database.

Note that this code was revised to store information directly into the sqlite database, instead of text files. It also archives the results of every successful response from the API. Meaning that the crawl can be canceled and resumed later without starting from scratch and hammering the API with repeated calls. Only the data in from before November 2014 was gathered with the functionally identical but clunky old version.

Sql database structure

We used the sqlite3 package with python 3.4.3 to construct and manipulate all of the sqlite databases. Copies of each database are available on request. The three original databases are Connery.db, Emerald.db and Miller.db. The data for a snapshot is recorded as a table of edges and edge attributes and a second table of node attributes. The table names are simply the month and day the scrape was run. So for example the snapshot recorded on the fourth of August 2014 would be saved in two tables 'Aug4e' and 'Aug4' containing

the edge set and node attributes respectively.

The source code is included in the code Appendix 8. Instructions for using the crawler are found in the comments.

However, looking at basic facts about how the network was changing over time was difficult using this raw 44 day data. For example an avatar created on August 8 who played for 15 minutes, made a single friend then never played again would appear in every snapshot until October 7th. This massive lag on changes makes it much harder to identify what is changing.

To gain a better insight we restricted the network to only those avatars who had logged in since the last snapshots was collected. This removed the massive lag on changes in the graph and it also reduced the size of the databases to a more computationally manageable level. We created the databases CW, EW, and MW to store this constrained sub network.

The following information is included in the avatar attribute table of each snapshot.

Id:

An avatar Id uniquely identifies an avatar.

name:

An avatars display name, unlike the Id names can be changed. For example offensive name are usually changed by the company to something more suitable.

br:

The avatars "battle rank" is the Planetside 2 term for avatar level, battle ranks start at 1 and are capped at 100. Unlike role playing video games Planetside 2 does not restrict an avatar based on level, instead new avatars are restricted by unlocking tools (weapons, explosives, vehicles, etc) or upgrades (explosive resistance, tougher armor, etc.) which are purchased with experience points. Level wikipedia

outfitTag:

A 2 to 4 letter identifier displayed in front of the avatars name for every outfit member (Outfit is the Planetside 2 term for a "clan" or "guild"). While each tag is unique, they may be changed. Avatars that are not currently in a outfit have "not available" or nothing instead of a tag. The inteconnections between outfits in Planetside 2 has been studied in [23] Clan wikipedia.

outfitId:

A unique 11 digit Id that denotes an outfit, unlike tags these cannot be changed.

outfitSize:

The total number of avatars in the outfit according to the in game roster includes large numbers of inactive avatars.

faction:

Each avatar belongs to one of the three warring factions the New

Conglomerate (NC), Terrain Republic (TR) and Vanu Sovereignty (VS) a faction is picked during avatar creation and cannot be changed. [4]

Creation_date:

A Unix time-stamp for the avatars creation.

Last_login_date:

A Unix time-stamp for the last login date and time of a avatar.

login_count and minutes_played:

The number of times this character has logged in and the lifetime minutes played.

kills:

The number of times this avatar killed an enemy avatar this month.

deaths:

The number of times this avatar died this month.

played_month:

The number minutes played this avatar was logged in for this month.

The statistics for kills, deaths and played_month were collected from historical records in the API. This means these values are the same for every snapshot taken in the same month. The code for doing this is included alongside in the source code for the crawler in Source 8 but must be run separately.

[4]Each faction has unique equipment and appearance though they function similarly.

Population

In this section we provide a overview of the basic attributes of the Planetside 2 snapshots and how they change over time. Including the population of avatars and the number of friendships among them in each snapshot. Along with the number of avatars are added and removed in each snapshot and how it changes.

We also start looking at the problem of node removal from a macro level and look at how many brand new avatars have been created, and the number of avatars who have gone inactive vs those who have been permanently abandoned.

It is important to keep in mind the distinction between players and avatars, the census API only provides information on individual avatars, player info is private. Thus one player can have many avatars and none of the information available through the API can directly identify which avatars correspond to the same player. Each account starts with 3 avatar slots, three more come with the purchase of a membership and a additional 3 slots can be bought separately, so a single account can potentially have up to 9 distinct avatars. Since Planetside 2 accounts are free it is possible for a single player to have more than 9 distinct avatars spread over multiple accounts.

Snapshot overview

Only the results for the CW, EW and MW are included here. The full 44 day networks are available on request.

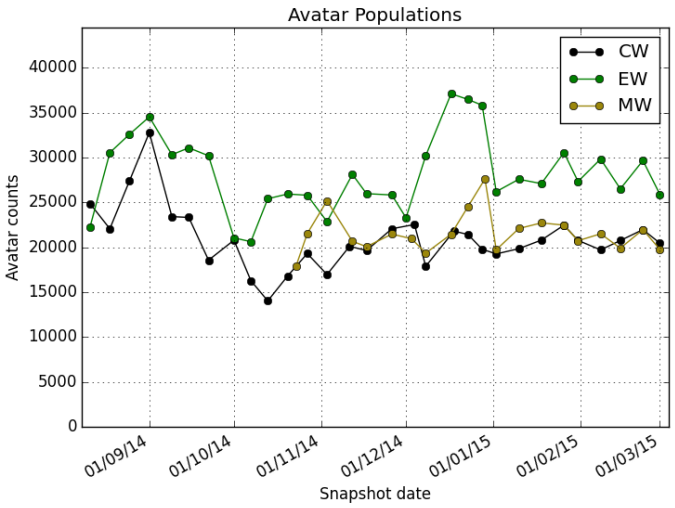


Figure 2.6: Avatar population

Figure 2.6 shows the total number of avatars per snapshot. The CW population collapse on Sept 15 and Oct 13 is the result of a database error that returned the last login date for about a quarter of all avatars as February first 1970. Figure2.7 shows the total number of edges per snapshot. The same error caused a reduction that can be seen in the Sept 15 and Oct 13 data. [5]

In terms of avatar population as shown in Figure 2.6, the newly merged

[5]Unfortunately this error obscures a global drop in population seemingly caused by the unpopular Halloween pumpkin patch.

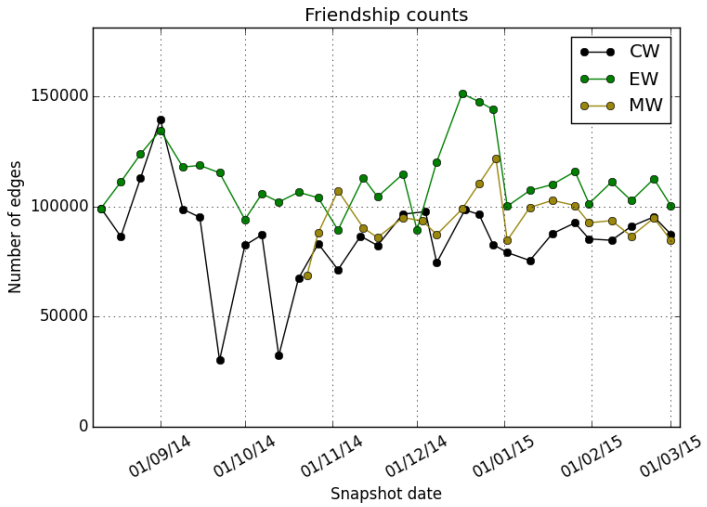


Figure 2.7: Total friendships

Emerald server has the highest population by a good margin and a growing lead on the other two servers. While Connery has the lowest and most unstable. All servers follow similar broad trend, after spiking in September the population declines through October to mid November then rises through December somewhat with another spike for the first month of the new year.

December signals a general increase in population for both Miller and Emerald with Emerald having the greatest uptake in avatars despite serious server issues specific to Emerald around that time. Once February begins that population immediately disappears. The opposite pattern holds on Connery where if anything population almost drops during the holidays.

Population change

In order to examine the changes in the population we have to categorize these nodes that have been removed.

Avatar states For the purpose of this work we say an avatar is active during a snapshot if it has logged in even once since the last snapshot. Likewise an inactive avatar is any previously seen avatar who is not currently active yet is active in future snapshots.

We say that a avatar is "new" if it was created between the previous snapshot was taken and the current snapshot. And we say an avatar is "dead" or abandoned if it is inactive and never becomes active in any future snapshot. It's also important to keep in mind that as we approach the end of our data the dead count increase because inactive avatars have less time to return.

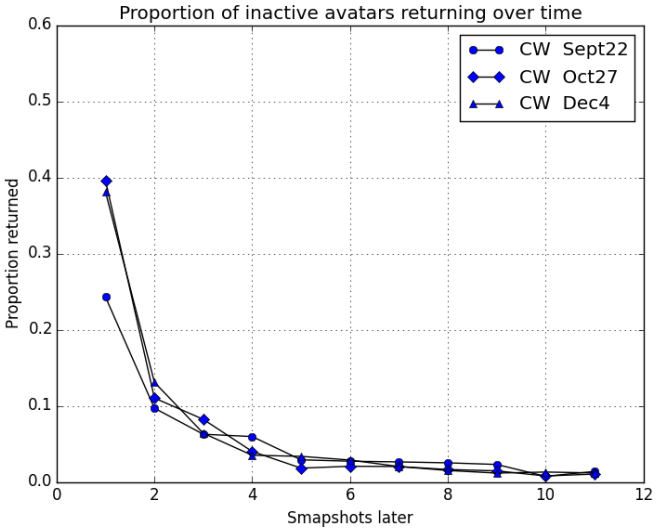
We also have a third class of new-dead or immediately abandoned (IA) avatars, those avatars are created and abandoned in the same snapshot, the intersection of the new and the dead avatars. This category is necessary because the new-dead make up more than half of the new in every snapshot.

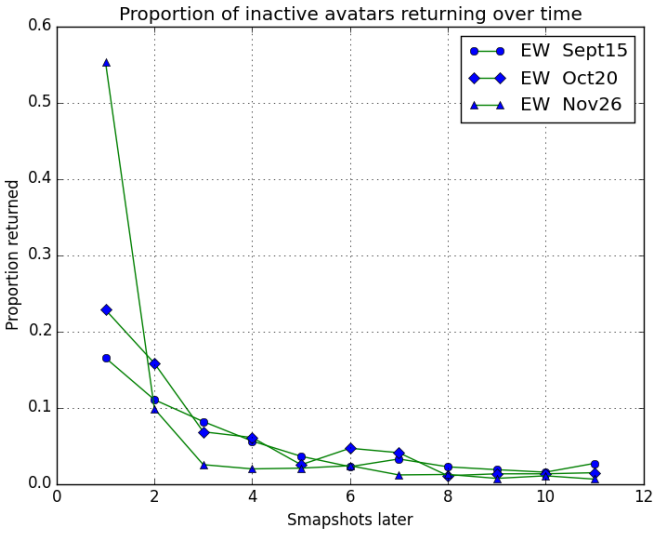
The meaning of new Choosing what constitutes a new avatar is not trivial. Depending on the definition the results can change drastically and analysis can be made much more difficult. Should we define new as new to

the game itself, or as new to socialization within the game, or new to the current social context. We use the first case in this paper by defining new as being created in the current snapshot. In the second case we would define a avatar as new when it forms its first edge to another avatar. The third case is exceptionally hard to detect and would require comparison of their neighborhood both before and after periods of extended inactivity. But it avoids the problem where very old inactive avatars returning from extended inactivity being miss-categorized.

Returning from inactivity

First we need to confirm that the implicit assumption that we can call an avatar dead if it goes inactive for an extended period of time. To do this we look at three snapshots from each server and see how many of the avatars who went inactive in each have returned in subsequent snapshots. The results are shown in Figure2.5. We see that the proportion of abandoned avatars returning as the weeks pass plummets from 40 % in the first week then 20% to less than 10% before eventually stabilizing at 5%. Notice that later snapshots see progressively less inactive avatars return.


Proportion of inactive avatars returning over time
CW Sept22
CW Oct27
CW Dec4
Proportion returned
Smapshots later
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
2
4
6
8
10
12


Proportion of inactive avatars returning over time
EW Sept15
EW Oct20
EW Nov26
Proportion returned
Smapshots later
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0
2
4
6
8
10
12

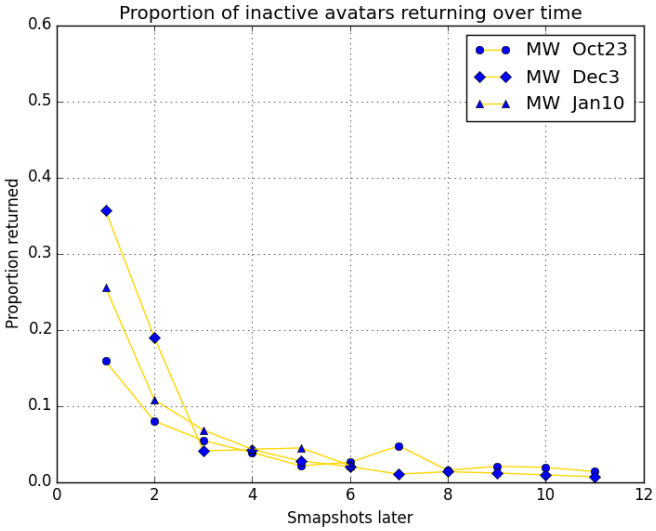


Figure 2.5: Inactive avatars who returned each snapshot.

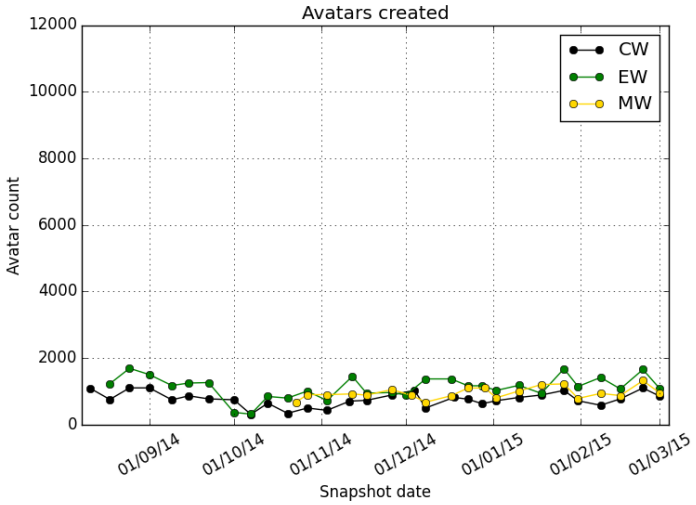


Figure 2.6: New avatars found in each snapshot.

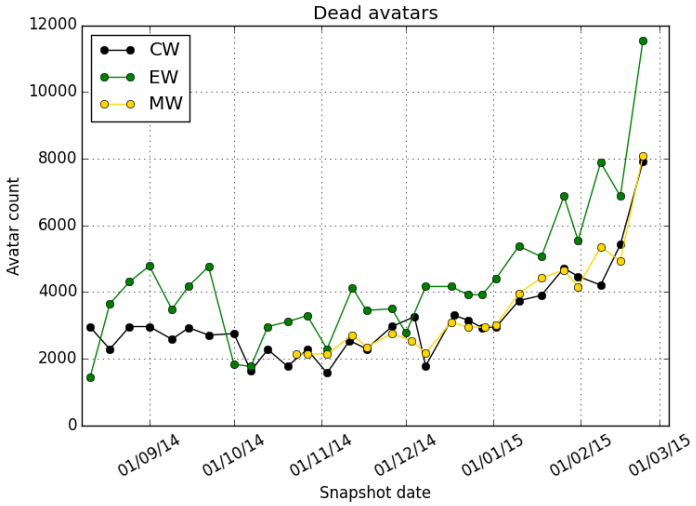


Figure 2.7: Dead avatars found in each snapshot.

In Figure 2.6 the first emerald snapshot is omitted due to a database error for creation date values.

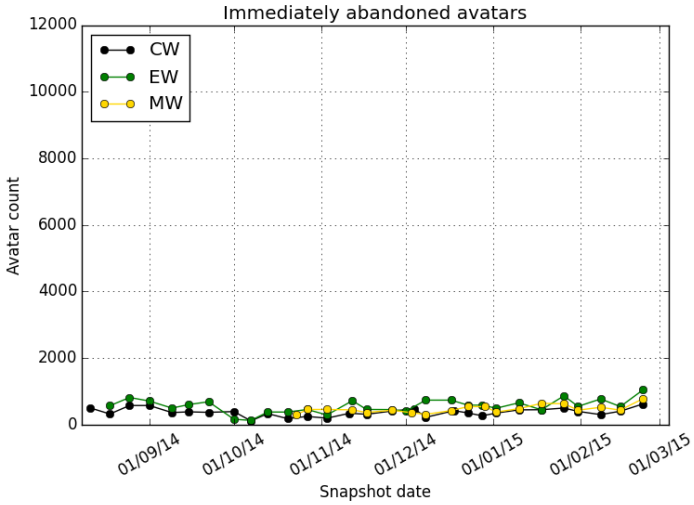
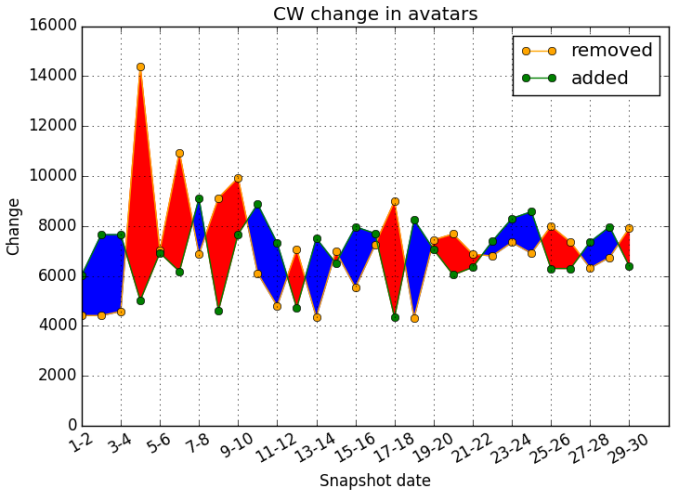


Figure 2.8: Immediately abandoned avatars by snapshot.



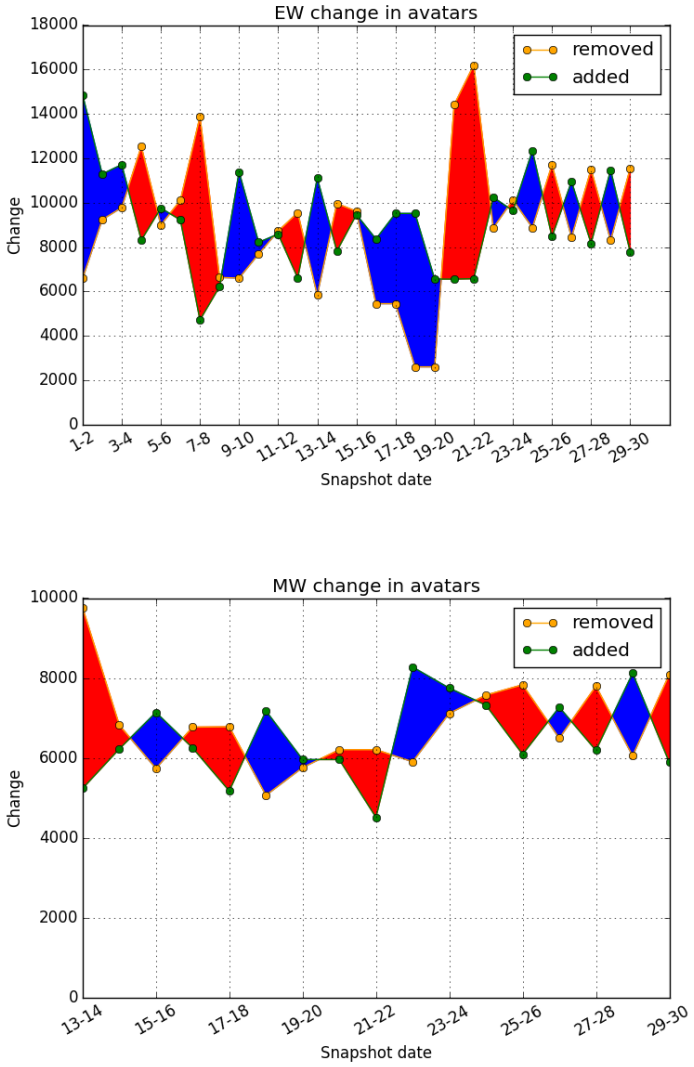


Figure 2.7: The difference in populations of consecutive snapshots.

The change in the population of avatars from one snapshot to the next for each of the three servers can be seen in Figure 2.7. The removed line indicates how many avatars went inactive while added indicates how many reactivated

or were created. These measures are more erratic and are considerably larger than the number of new and dead avatars found in each snapshot. We can see the main drivers of the population drop in the fall was avatars going inactive, who subsequently returning resulting in the winter stabilization.

Population trends.

All servers have a gradually decreasing population and a proportional decrease in the number of friendships. As time passes the avatars who go inactive take longer and longer to return. Though they do eventually return since the number of avatars created and destroyed is much more stable. So the population decrease is mainly caused by avatars going inactive for increasingly long periods of time rather than quitting entirely. In fact a very large proportion of the population in any given snapshot consists of avatars created in the first weeks after the game was launched as shown in Figure 9.1.

Snapshot density and average clustering coefficient.

The average clustering coefficient and density of each snapshot are shown in Figure 2.8 note that the density values are multiplied by 100 to allow comparison on the same scale. Clearly the Average clustering coefficient of the Planetside 2 dataset is much much larger than the density, meaning that these networks exhibit high clustering among its nodes as expected in a social

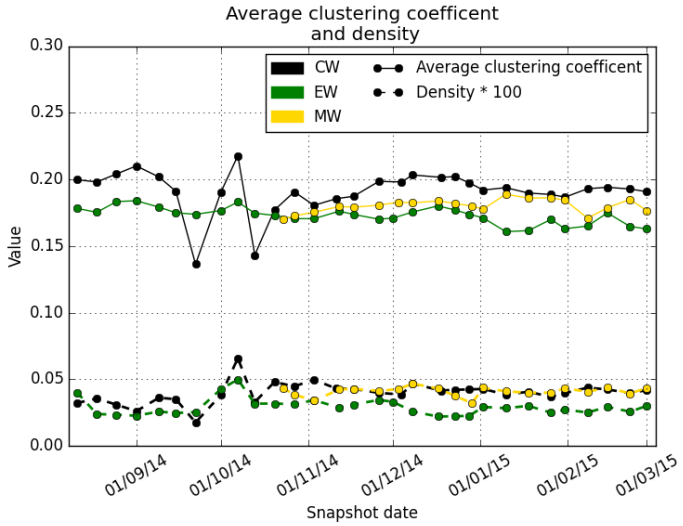


Figure 2.8: Snapshot density and average clustering coefficient

network [16].

Small world property

Planetside 2 snapshot diameter and APL

Figures 2.9 2.10 2.9 show the diameter and APL of the giant component for every server. The snapshots are too large to compute the diameter and APL of the whole snapshot in a reasonable amount of time, so we computed values for each faction individually. We also computed the values for the combined the TR and NC factions to get an idea of how much greater the

values of the whole graph would be. [6]

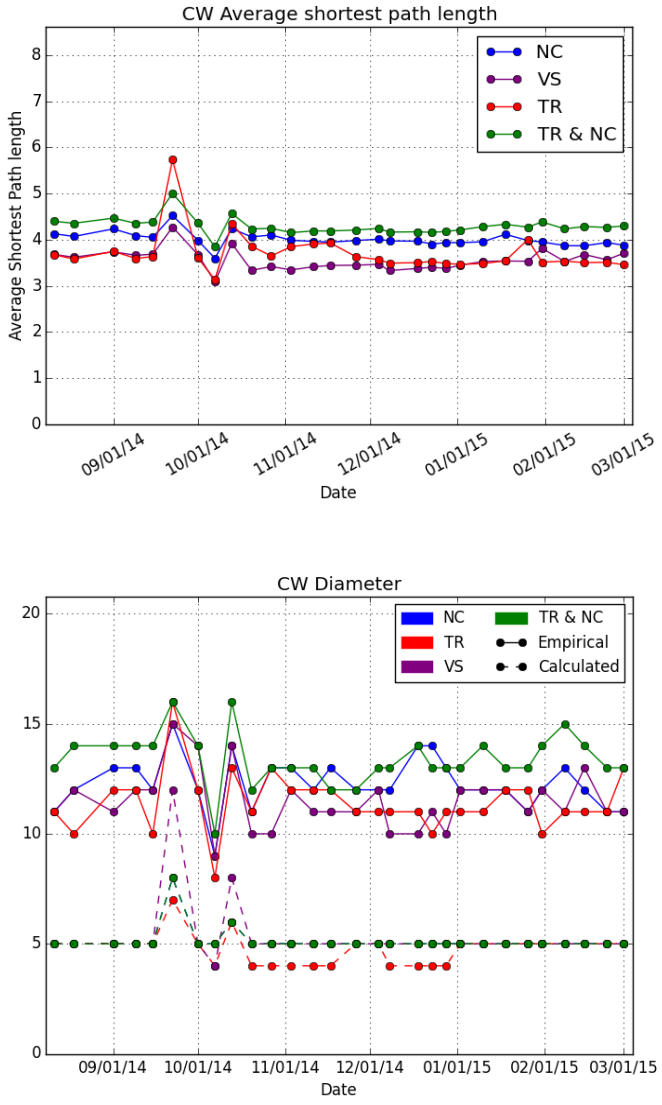


Figure 2.9: CW diameter and average path length.

[6]The factions with the highest APL is always the the faction which has a reputation for being disorganized on that server. All three scored the least victories during the time of data collection.

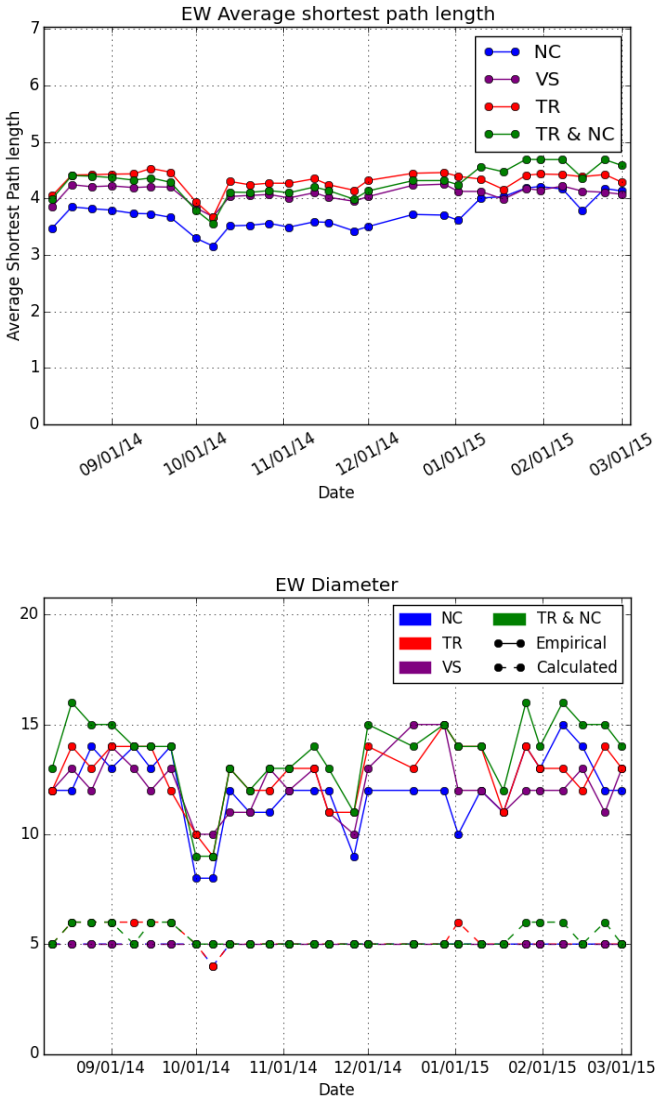
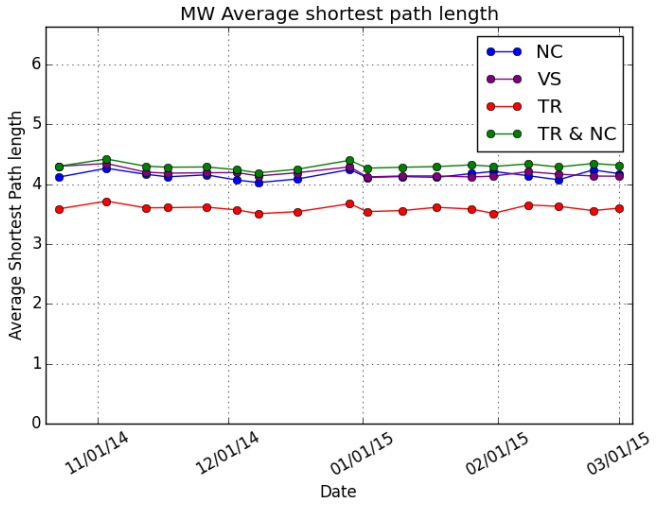


Figure 2.10: EW diameter and average path length.

When we look at Figure 2.10 we see an odd phenomena in APL from the beginning of the data until January the combined TR & NC faction actually have a lower APL than the TR faction alone, meaning that for TR running

through a NC avatar is often the fastest way to reach fellow TR avatars. This is caused entirely by a single avatar, Klypto. In January and Febuary Klypto only logged in during the week of February 15th and we clearly see a corresponding impact on the APL NC of the combined NC and TR & NC. [7]

This was easily confirmed by removing Klypto from earlier snapshot and computing the APL. This avatar might also have an impact on diameter but is not as clear and fails to show up when Klypto is removed in most earlier snapshots. [8]



[7] The official website times before it can successfully load his friend list, allegedly this player sends a friend invite to everyone he sees.

[8] The APL of the VS is not affected by Klypto in the same way.

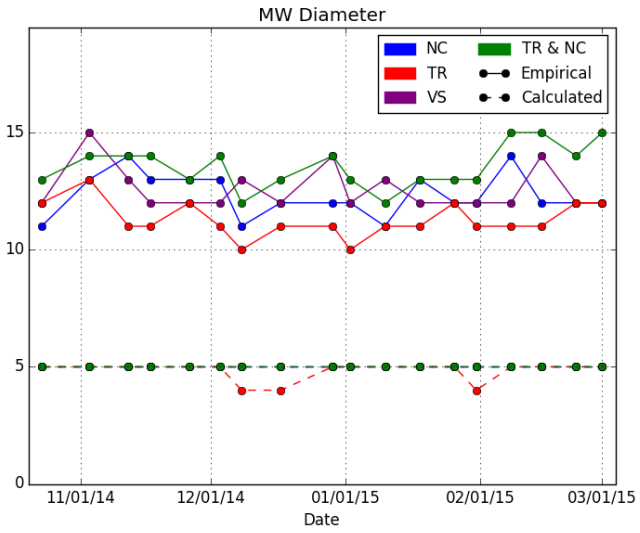


Figure 2.9: MW diameter and average path length.

As we can see the APL in our real data set is lower than these the upper bound expected $G_{n,p}$ random graph values. While the diameter is significantly greater tha the bound. Thus the friendship network has large diameters and short APL. Since the majority of avatars are closely bound to a well connected core but there are many low degree avatars at the end of long chains of other low degree avatars that at best tenuously connected to the giant component at all.

Thus the Planetside 2 dataset immediately fails to be small world not meeting the first criteria for a small world graph.

Planetside 2 clustering coefficient

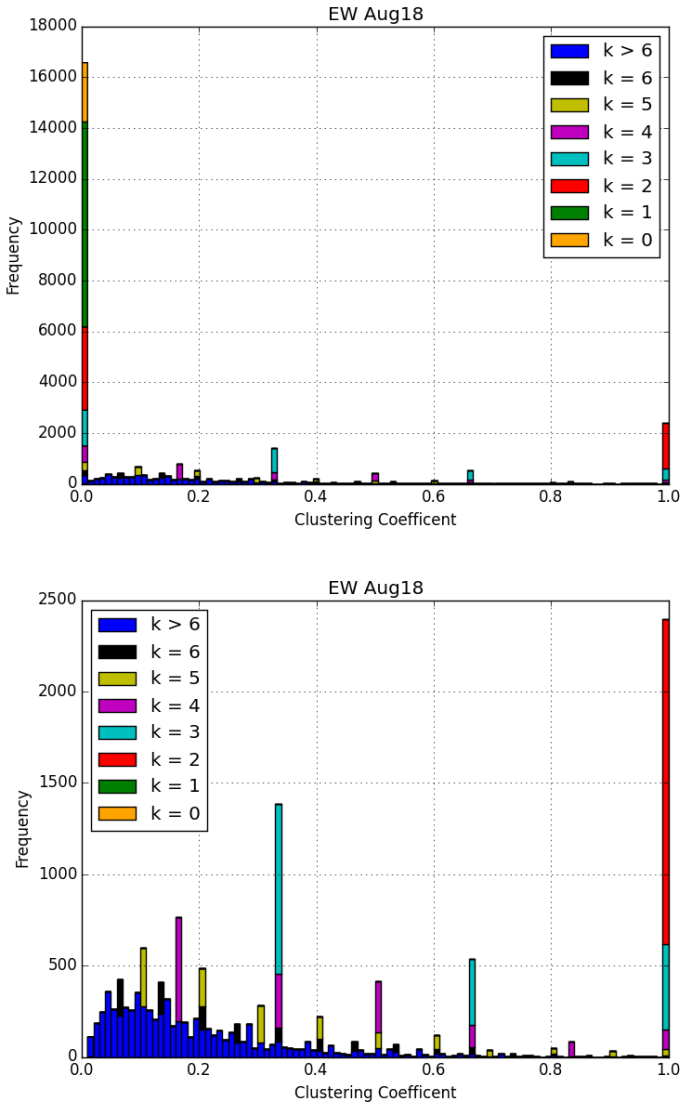


Figure 2.10: The clustering coefficient distribution with and without the spike.

An example of a distribution of clustering coefficients can be seen in Figure 2.10 from the emerald server on the 18th of August. The majority of avatars have a clustering coefficient of zero, because of the prevalence of avatars with only one or two friends. The lesser spikes are a result of the limited number of values for low degree avatars and abundance of such avatars. For example a degree 3 avatar has three potential mutual friends so the clustering coefficient can be 1, 2/3, 1/3 or 0 and as we can see there are spike consisting of degree 3 avatars at 0 , 1/3 , 2/3 and 1, a degree 4 avatar has 7 possible coefficients a degree 5 has 21 and so on.

Overall the avatars have a clear bias towards low clustering coefficients which is also reflected in the individual degree categories. Though this is still far more clustering than expected from a random network.

Planetside 2 degree distribution

Now we determine if the degree distribution of the snapshots follows a power law, using the methods as before. A typical degree distribution can be seen in Figure 2.11 this one is from CW on Dec 17th.

The results are summarized in Table 2.4 the values for each individual snapshot can be found in the Tables 9.2 in the additional Figures appendix.

As with the competition network the log likelihood test indicates that the power law model is superior to the exponential model. While the exponential

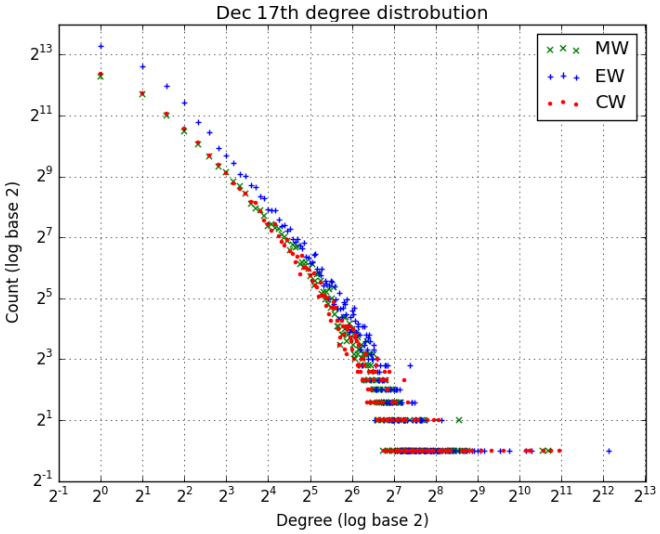


Figure 2.11: Typical snapshot degree distribution

				$x^{-\alpha}$ vs $e^{-\lambda x}$	$x^{-\alpha}$ vs $x^{-\alpha}e^{-\lambda x}$	
Date	xmin	D	α	(R,p)	(R,p)	n tail
				CW		
Mean	97.370	0.040	2.203	(3.184,0.0017)	(-0.7634,0.1205)	126.704
SD	8.996	0.007	0.0491	(0.1704,0.0011)	(0.1835,0.1205)	55.728
				EW		
Mean	117.964	0.052	2.237	(3.0933,0.0043)	(-0.4050,0.7787)	163.786
SD	46.713	0.008	0.074	(0.4346,0.0127)	(0.1288,0.1234)	38.686
				MW		
Mean	106.412	0.038	2.342	(3.0556,0.0023)	(-0.3185,0.0703)	179.529
SD	12.057	0.004	0.044	(0.0878,0.0007)	(0.0922,0.0703)	57.755

Table 2.4: Summarized power law results for snapshots.

cutoff model is superior to the pure power law model but only has significance until xmin is reduced to around half its optimal value. The value of the xmin across all snapshots was stable at around 100. While the number of avatars that were in the tail varied according to the overall trends in the population.

There are three snapshots that are worth mentioning individually, in the EW snapshots on Dec17 Jan 10th and Jan 31 each had actually xmin is nearly 300. In these snapshots the exponential cutoff model is a better fit with significance for any value of xmin.

Figure 2.12 is an example of the minimization of D for values of xmin and as is typical when xmin approaches 1 the value the quality of the best power law fit decreases.

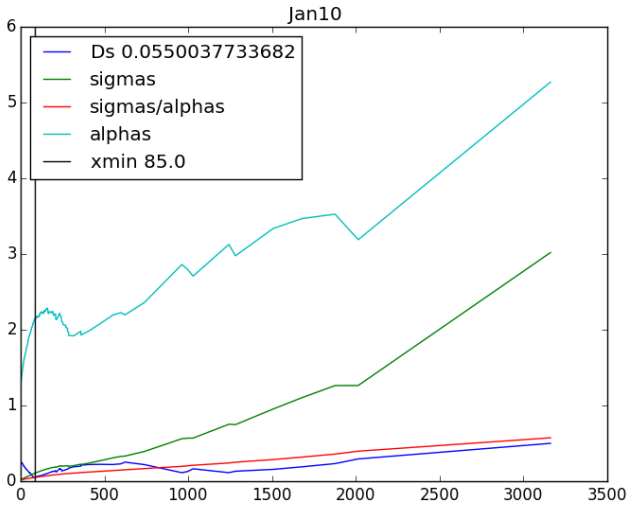


Figure 2.12: The alpha and sigma (error on alpha) for all values of xmin.

When we use xmin lower than the optimal the power law with exponential cutoff model becomes a better choice with significance just like the competition networks. So in conclusion the Planetside 2 datasets are also not true examples of small world networks as a result of their large diameters.

Chapter 3

PlanetSide 2: Preliminary Analysis

This chapter covers how relationships between vertex attributes change over time for both mixing patterns and the avatar attributes in the PlanetSide 2 snapshots. Assortativity

Patterns in assortativity

Assortativity as described in [22] is the tendency of nodes to be connected to nodes similar or dissimilar to themselves in some way. The assortativity coefficient reflects how strongly the members of groups tend to stick to their own.

The assortativity coefficient is defined as follows:

First let $e_{i,j}$ be the fraction of edges that connect vertexes of type i to vertexes of type j in a graph G. Let $\mathbf{e}$ be the assortativity matrix of $e_{i,j}$ elements. In the case of a directed graph, a_i to be the fraction of outgoing edges connecting to a vertex of type i and b_i to be the fraction of incoming edges connecting to a vertex of type i. In undirected graphs like ours $a_i = b_i$.

Finally the assortativity coefficient is defined as

$$r = \frac{\sum_i e_{i,i} - \sum_i a_i b_i}{1 - \sum_i a_i b_i}$$

.

For attribute assortativity it is easy to compute the minimum value since a perfectly disassortative partition would have no edges between members of the same partition thus the entries along the diagonal of $\mathbf{e}$ would all be zero thus the minimum is simply:

$$r_{min} = \frac{-\sum_i a_i b_i}{1 - \sum_i a_i b_i}$$

.

We tested the degree assortativity and compare it to other online networks. We also examine assortativity by outfit membership, and relative skill in first person shooters the kill to death ratio. Then finally examine the assortativity for on battle rank.

Results

These Figures 3.3 3.1 3.4 3.5 3.2 show the assortativity coefficients of each snapshot in each weekly database. The calculations were done with networkX implementations based on [22].

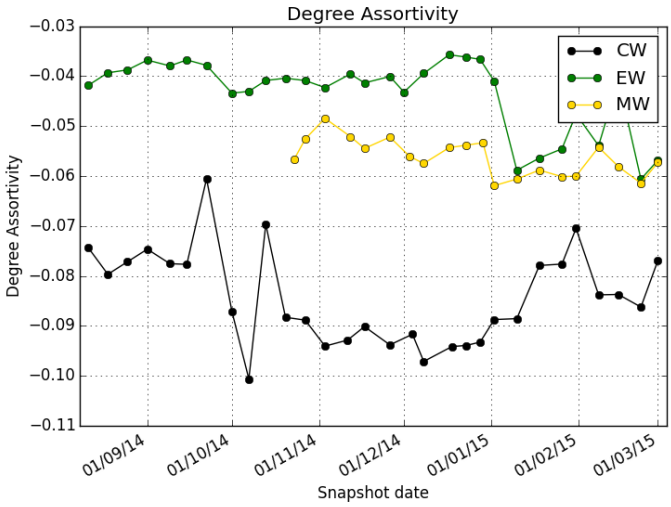


Figure 3.1: The degree assortativity of all snapshots by snapshot.

Initially it was hypothesized that a positive degree assortativity was a property of human social networks [22] while structural networks display a negative degree assortativity. More recent work [14] have found that this is not true for all social networks especially online where social networks tend to have a low or negative degree assortativity. Other studies of MMOG social networks [28] also found it to be negative. The degree assortativity of the Google network was -0.13 while Bing had -0.17 for instance.

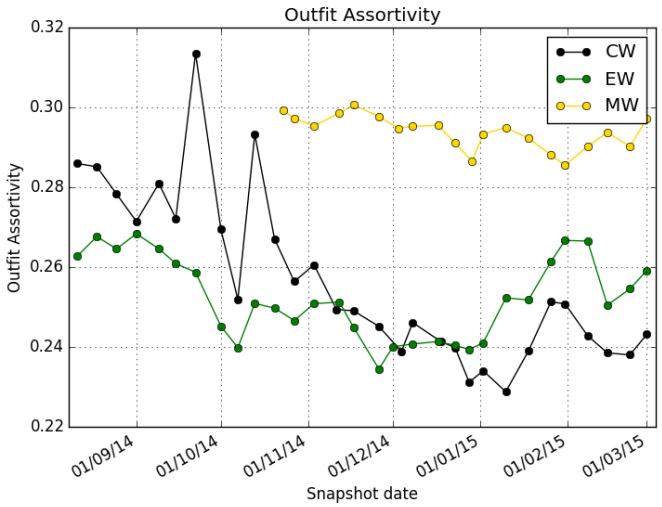


Figure 3.2: This assortativity of outfit members by snapshot.

In Figure 3.2 we see the second strongest assortativity is between outfits members at around 0.3 and as before we have similar values across all servers. Interestingly despite the fact that around 30% of avatars belong to no outfit, removing them from the network only increases the outfit assortativity slightly and as time passes the assortativity between outfit members degrades remarkably, possibly as a result of the changing population.

In Figure 3.3 we see that as time passes battle rank assortativity becomes stronger, this is probably a result of max level avatars building up. Note that the first two spikes in the CW line are a result of the previously mentioned database error.

As we see in Figure 3.4 the faction assortativity is very high around 0.9

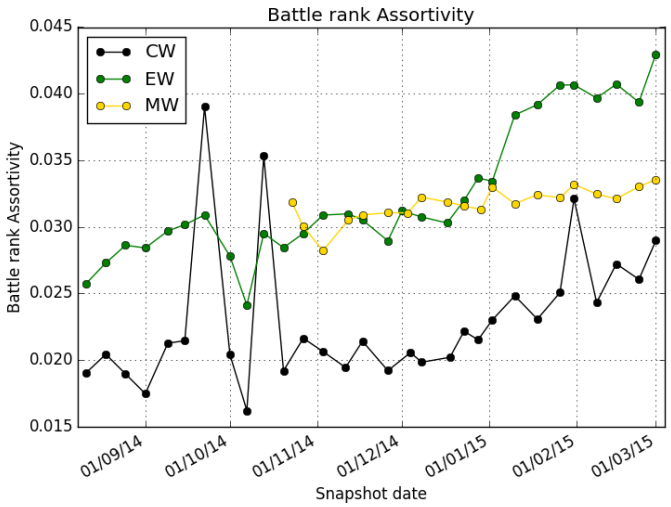


Figure 3.3: Battle rank assortativity by snapshot.

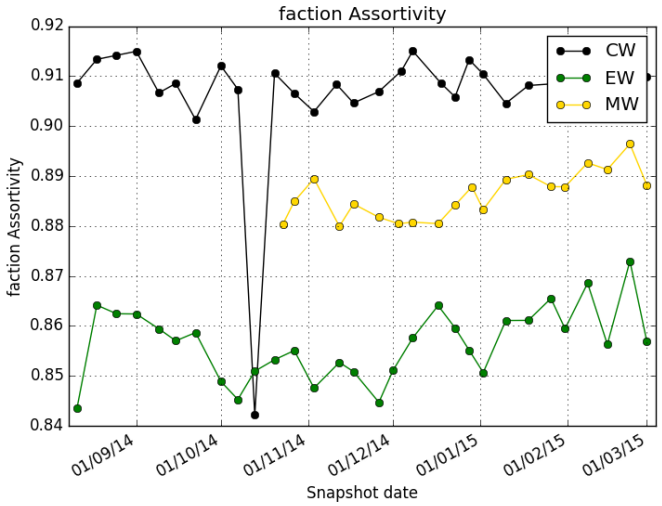


Figure 3.4: Faction assortativity by snapshot.

because unsurprisingly most avatars stick with members of their own faction.

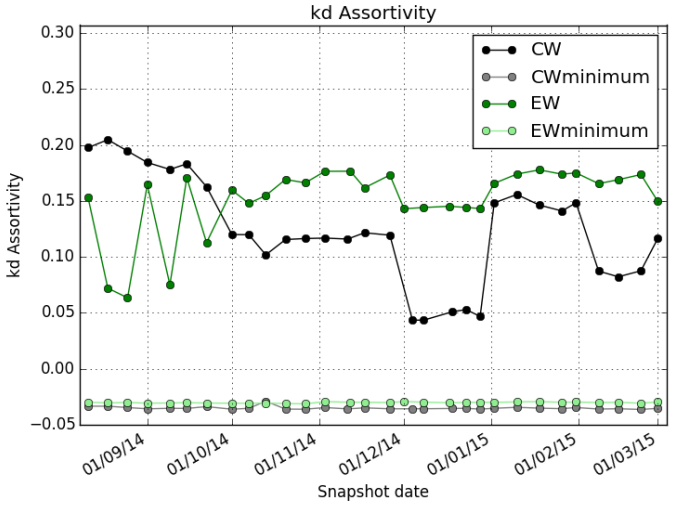


Figure 3.5: Kill/Death assortativity by snapshot.

Figure 3.5 shows the assortativity of the kill to death ratio by month [1] of the avatars in each snapshot from EW and CW. This figure also includes the lower bound for the assortativity. It was constructed using only avatars with more than 10 kills and 10 deaths, rounded to the nearest tenth to reduce the numer of groups. Even though kills and deaths are monthly values weekly variations are still seen because the active population itself varies.

It is interesting that those avatars with high degree actually prefer to link to others with lower degree, one of the common structures we see in this

[1]The kill to death ratio is simply number of kills over the number of deaths that month, and is the standard (if flawed) metric for skill.

network was high degree avatars with a halo of hundreds of degree one friends. Typically this group of avatars are new rather then returning low degree veterans. Supporting the idea that the some hubs in the planetside network actively seek out new avatars that otherwise have would stay isolated.

This is not surprising given that as avatars are balanced [2] having more people is often the path to victory. So links to other players is a benefit to a leader, and leaders must compete with each other for a finite pool of troops. Which also explains why these hubs are friends with nearly everyone except each other.

So in general the players of Planetside 2 like humans in other social contexts prefer to associate within whatever they consider to be their community. Which in this context means outfit and faction. The fact that so few of them seem to about individual skill or time played was not expected.

Node attribute correlations

In this section we take a look at the relationships between the various avatar attributes we gathered. By looking at the correlations between our numeric avatar attributes and graph attributes and patterns in the creation and deletion of edges and nodes. We discovered that many of these attributes are redundant for our purpose of identifying patterns in the removal of avatars.

[2]Have roughly equivalent power.

Correlation matrices

The correlations between avatar attributes are stable over time so for brevity we only include the average value in Table 3.7. The mean correlation matrices were constructed from the average of the correlation matrix of every 4th snapshot. Note that Miller was not included due to errors in its historical data that prevented recovery of kills deaths and time played.

Mean Pearson r correlation									
	degree	clustering	centrality	br	kills	deaths	kd	time	outfitSize
degree	1.000								
clustering	-0.016	1.000							
centrality	0.825	0.031	1.000						
br	0.312	0.024	0.379	1.000					
kills	0.250	-0.004	0.307	0.507	1.000				
deaths	0.205	-0.013	0.223	0.437	0.789	1.000			
kd	0.112	0.001	0.155	0.345	0.287	0.173	1.000		
time	0.243	-0.010	0.261	0.506	0.788	0.883	0.242	1.000	
outfitSize	-0.026	-0.085	-0.051	0.039	-0.008	0.047	-0.009	0.035	1.000

Figure 3.6: Average attribute correlation matrix for EW.

Looking at the matrices the strongest correlations are between kills, deaths and time played per month. Meanwhile degree and centrality are also correlated as expected since this network is very strongly dominated by its hubs. Notice that time played correlates more strongly with deaths than with kills. Since playing more guaranties dying more but does mean succeeding more. Battle rank likewise correlates to kills more strongly than deaths.

So the more a avatar is played and consequently the more kills, deaths,

Mean Pearson r correlation									
	degree	clustering	centrality	br	kills	deaths	kd	time	outfitSize
degree	1.000								
clustering	-0.005	1.000							
centrality	0.612	0.079	1.000						
br	0.305	0.059	0.293	1.000					
kills	0.210	0.006	0.167	0.499	1.000				
deaths	0.206	0.001	0.166	0.445	0.794	1.000			
kd	0.103	0.004	0.098	0.403	0.324	0.194	1.000		
time	0.246	0.004	0.193	0.510	0.792	0.892	0.280	1.000	
outfitSize	0.024	-0.033	0.020	0.088	0.003	0.065	0.008	0.056	1.000

Figure 3.7: Average attribute correlation matrix for CW.

and battle ranks they accrue the more friends they will tend to have; likely because they meet more avatars. Thus they have a lower clustering coefficient, since as degree grows clustering coefficient naturally decreases [26].

While our avatar attributes can tell us a lot about that avatar when it comes to avatar death, kills deaths level and time played are all measures of the same thing; commitment modified by skill at first person shooters and play style. While the exact values of these attributes can tell us a lot about an avatar like what role [12] to preform within the context of the game. For example a low time but higher Br with a typical kills and death numbers might indicate that avatar is a paying member with boosted exp income. While a avatar with a lower br, kills but large time played might indicate a player who prefers to socialize rather than fight, or indicate an avatar who specializes in transport/logistics. [3]

[3]There is a widespread belief among Planetside 2 players that members of larger outfits

are less individually skillful. This is not supported by the data.

Chapter 4

Node Expiration

This chapter covers the pattern we found in the removal of nodes from both data sets. First starting with the relation node removal and simple graph properties like degree and clustering coefficient. In part two we examine the correlations between the node attributes we have available and examine their impact on node removal rates.

Competition network

For the competition network we look at the patterns in the degrees clustering coefficients, average neighbor degree as well the edges connecting them and the components of the induced sub graph of dead companies. We found that 2.7% of all domain names in our data were available for purchase.

See Table 4.1.

	Bing	Google
Companies	279	607
Edges	51	101

Table 4.1: The failed companies and edges connecting them.

Of these dead companies all 101 edges in the Google network had weight 1, while in Bing there were 4 edges of weight 3 and one of weight 5 with the remaining having weight 1. So we decided to compare the number of edges among the failed companies to the number in a random subset of the same size, which gives us the boxplot in Figure 4.1 which compares the number of edges connecting in the subgraph induced by the failed companies to the expected number of edges for random subgraphs of that size. The red dot is the number connecting actual failed companies, clearly in both networks these companies have far fewer connections than we would expect, meaning the failed companies are more isolated from each other.

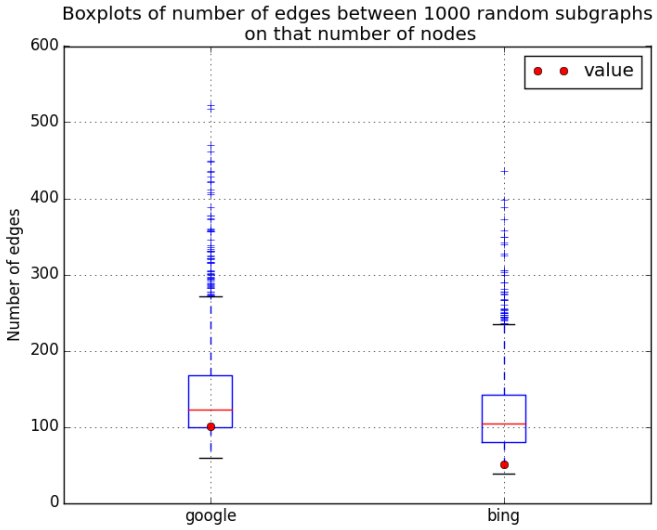


Figure 4.1: Edges in random subgraphs.

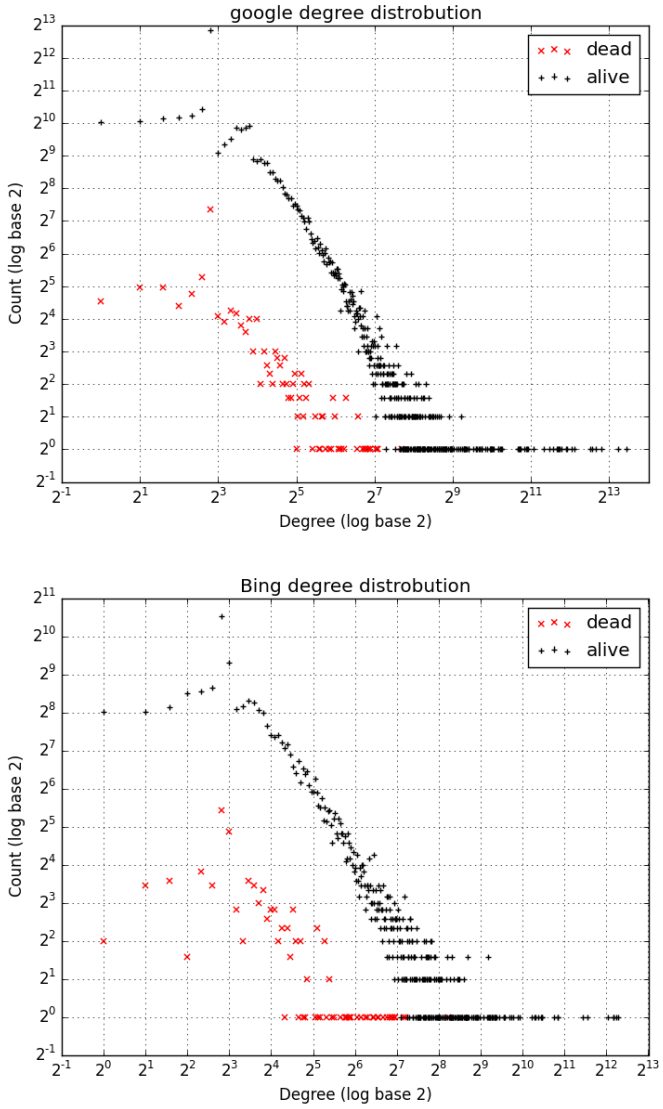


Figure 4.2: Degree distributions in the competition networks

When we compare the degrees of the failed companies to those of the network as a whole as in Figure 4.2. As we can see the trend for the failed companies is consistent across both advertisement networks.

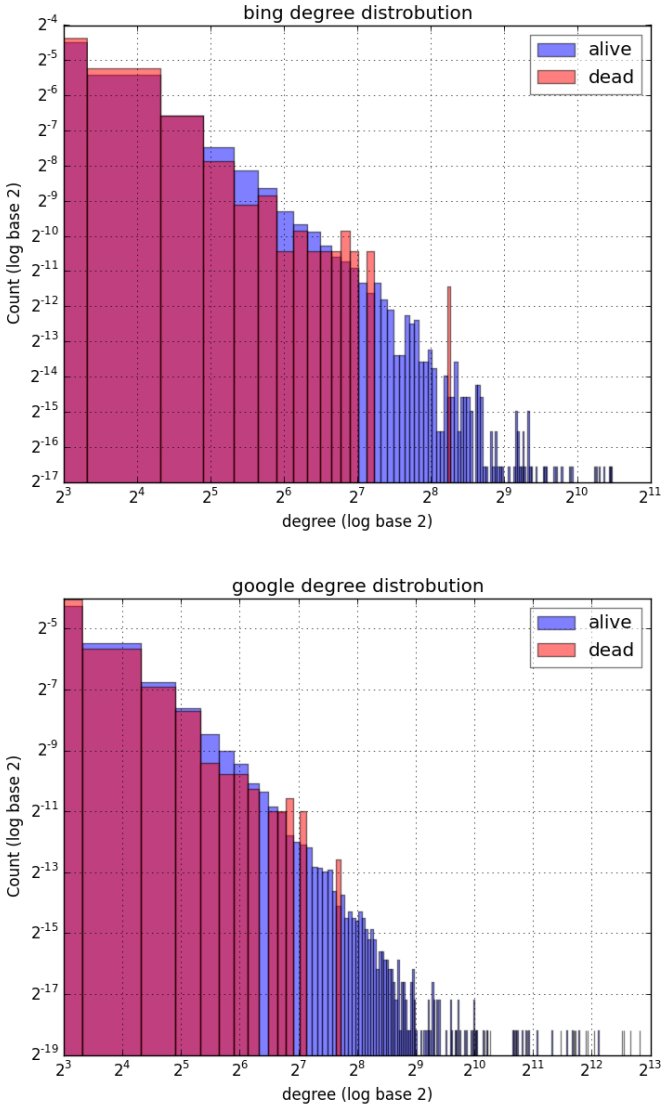


Figure 4.3: Degree distributions in the competition networks (log bins)

Figure 4.3 also shows the degree distribution but using logarithmic bins. This more clearly shows that patterns are consistent across both advertisement networks. Though none of the highest degree companies failed the distribution

of failed companies follows the global distribution.

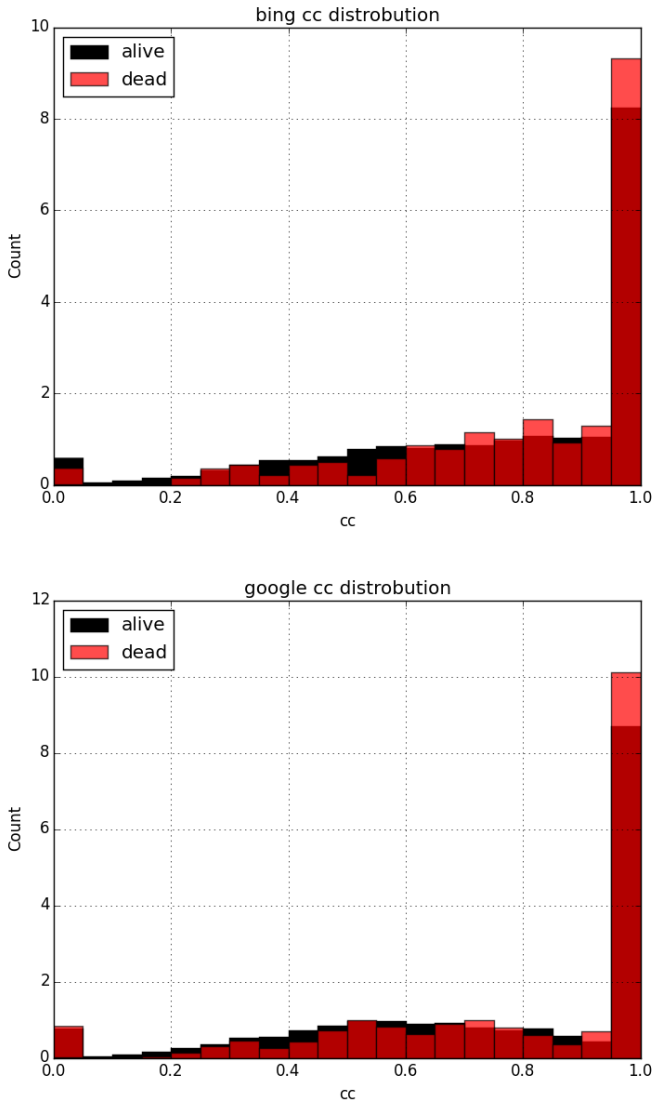


Figure 4.4: Clustering coefficients distribution for failed companies.

Neither the degree distribution in Figure 4.3 or clustering coefficient distribution in Figure 4.4 of the removed companies deviate from the global

tendency too much. There is a slight bias towards higher clustering and lower degree. But since many of the companies on the high end of the degree distribution are places like Google, Ebay and Amazon who are unlikely to close and whose domains would never be allowed to lapse even if they did.

Communities of failure

Table 4.2 shows the number of components of each size in the network of failed companies; both networks have a single 13 company component but they are not the same 13 companies.

Size	1	2	3	4	5	6	7	8	9	10	11	12	13
Bing	212	14	1	1	1	1	0	1	0	0	0	0	1
Google	472	31	9	3	2	1	1	0	0	0	0	0	1

Table 4.2: Failed components of the competition network

One of the first things we searched for was clusters of failed businesses, by comparing the subgraph induced by the failed businesses to subgraphs induced by the same number of randomly selected businesses we get boxplot shown in Figure 4.1. We see that the failed business data had fewer connections than expected subgraph of its size. Those failed companies that were adjacent appear to be the result of market failures rather than a result of the network structure.

For example in the Google network the 13 companies in the largest connected component are related to the home mortgages and realty, these were also the highest degree failed companies. Their failures likely have more to do with the collapse of the housing market than any aspect of our competition network. The second largest was consisted entirely weight loss/supplements, while the third was homeopathic medicine. There was also a component of 9 selling mobility installations for the elderly, which also probably a result of the housing bust.

Things are similar in the Bing network the largest component consists of realty and home mortgage companies including 5 from the largest Google component, as well as a few companies with no information. The second largest component includes a large component of 7 homeopathic medicine / weight loss and the third consists mostly of supplements. Another example of removed nodes common to both networks is a group of 3 connected moving companies, however in Google are connected to realty sites component but not in Bing. [1]

Competition network conclusions

Since many of the companies involved in these large components seem to be evidence for companies failing as a result of the market contracting.

[1] It also includes a group of three websites selling wood working patterns which folded after the source of woodworking patterns were released for free online.

For the large majority of business failure is a isolated event, where a single company failing does not impact their neighbors likelihood of failure. Since one competitor dropping out should make business easier for there former competitors. In conclusion those companies that fail are typically in peripheral with unremarkable structural properties. And they will typically not be adjacent to one another unless they experience market.

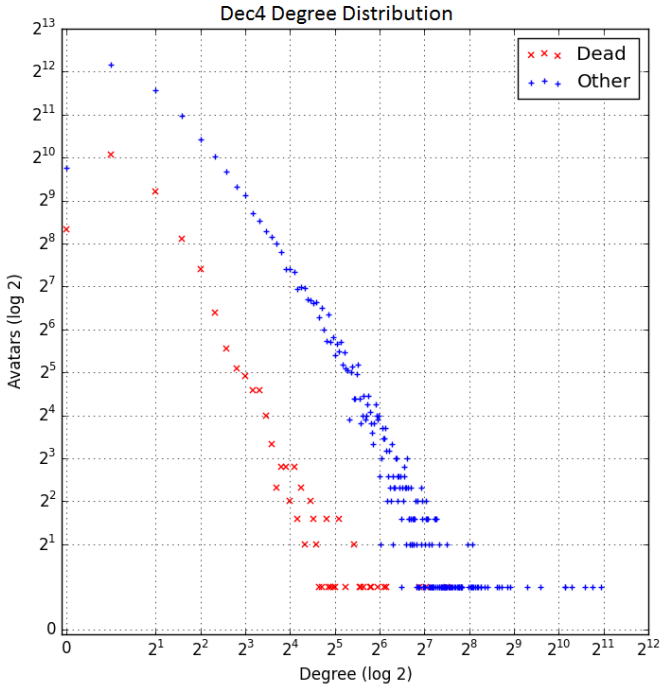
PlanetSide 2 avatar and friendship removal over time

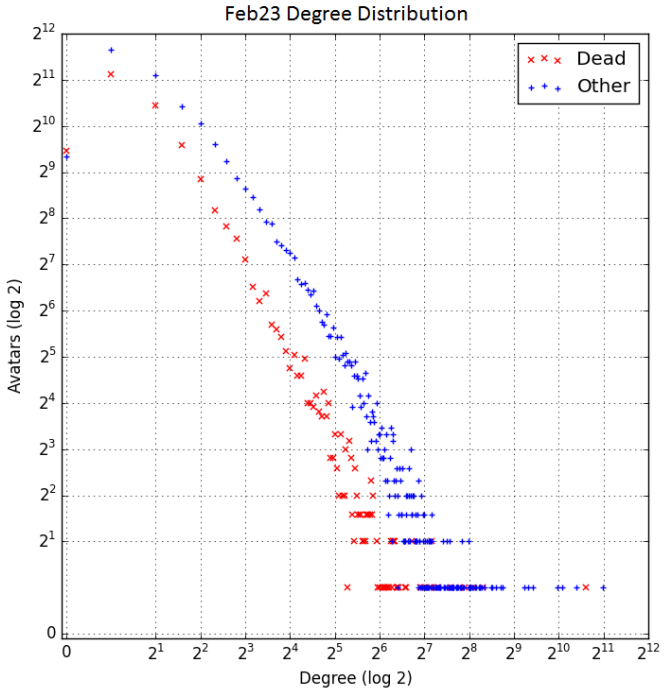
Since the avatars of Planetside 2 have a high removal rate its analysis can be more in depth and including dynamics and edges.

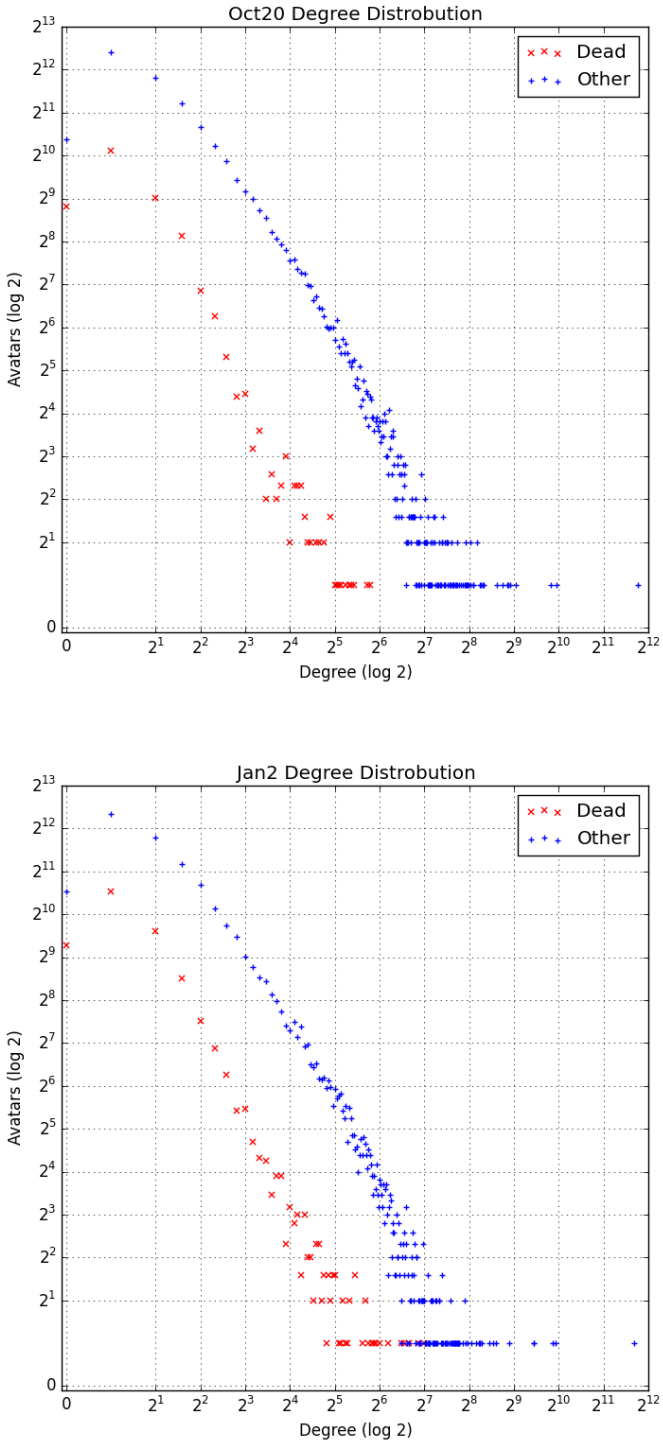
Basic trends

Degree

This section covers the effect of the degree distribution on the removal of avatars in order to confirm that low degree avatars are more likely to be removed.


Dec4 Degree Distribution
Dead
Other
Avatars (log 2)
Degree (log 2)


Feb23 Degree Distribution
Dead
Other
Avatars (log 2)
Degree (log 2)



66

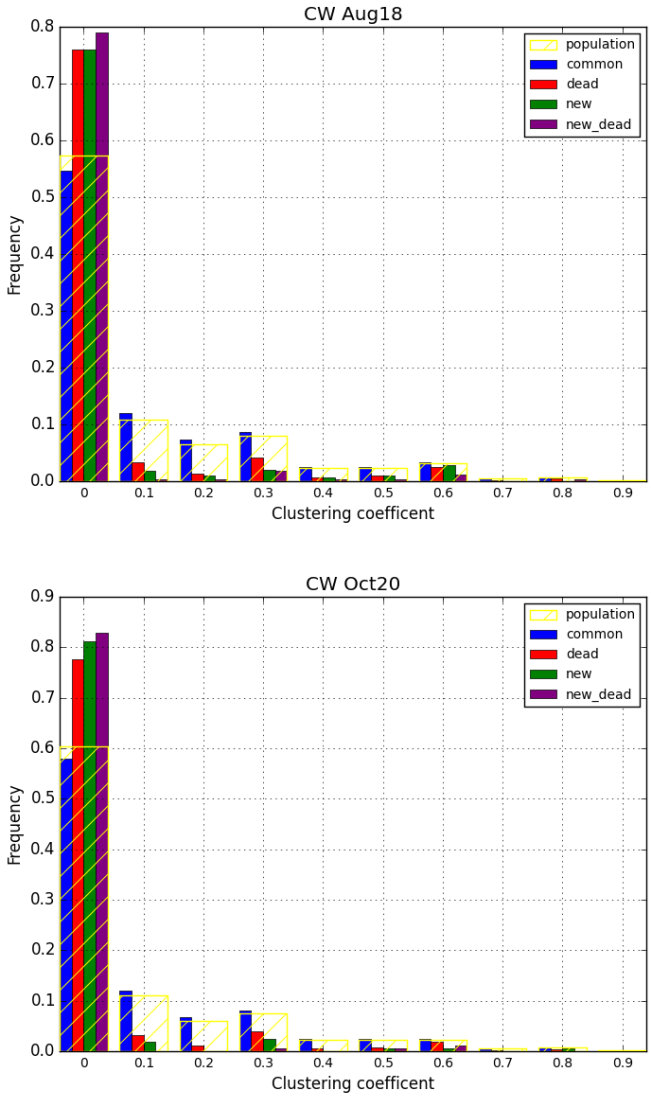
Figure 4.5: Avatar death by degree.

And as we can see in Figure 4.5 the removed avatars tend to be lower degree than the others.

Clustering coefficient

In this section we examine the clustering coefficients of avatars by state. We test the intuition that avatars with a higher clustering coefficient will naturally stay active longer.

Figure 4.6: CW clustering coefficients.



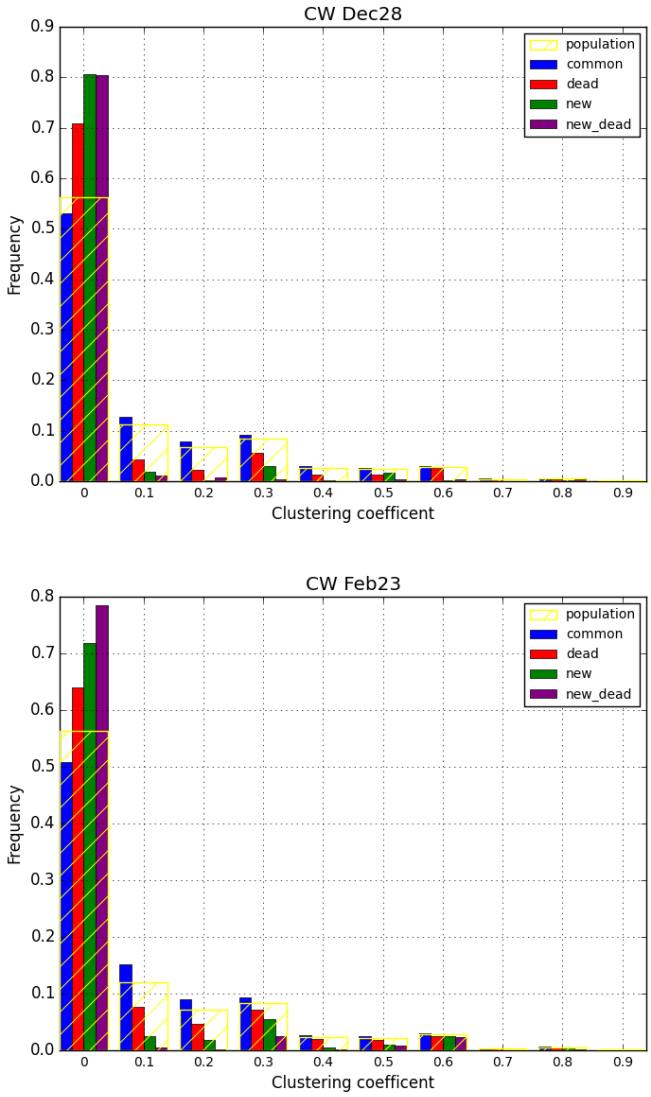


Figure 4.5: Avatar death by clustering coefficient.

Figure 4.5 shows the clustering distribution of avatar by state normalized for population. More examples are available in the appendix 9.5. Compared to our other measures the clustering coefficient is irregular because avatars

with clustering coefficients from 0.5 to 1 are rare combined with the relatively small numbers of dead new and immediately abandoned (IA) avatars results in empty bins. However it is clear that a higher clustering coefficient is better for survival.

Edge dynamics

In this section we take a look at the dynamics behind the creation and removal of edges and unstable relationships. By creating a graph from the combined edge sets of each snapshot, and using edge and node attributes to record the dynamics of each as they are added. The source code used to do this is included in the code Appendix 8. And the total number of Avatars and edges in this graph is shown in the first section of Table 4.3.

	CW	EW	MW
Basic Information			
Avatars	111655	151227	86178
Edges	397592	504413	313175
Avatar type breakdown			
New Avatars	10.89%	11.24%	11.79%
IA Avatars	9.80%	10.47%	10.45%
Dead Avatars	72.01%	72.53%	66.84%
Edge formation			
existing ↔ existing	29.97%	29.23%	25.15%
existing ↔ new	4.74%	5.27%	4.85%
existing ↔ IA	3.31%	3.71%	3.32%
new ↔ new	0.45%	0.53%	0.56%
new ↔ IA	0.30%	0.41%	0.39%
IA ↔ IA	0.23%	0.29%	0.30%
Edge deletion			
One removed	53.15%	54.70%	52.01%
Both removed	4.30%	5.02%	4.54%
Broken	4.06%	4.23%	4.69%
Unstable	0.29%	0.30%	0.26%

Table 4.3: Edge formation and deletion form each of the three servers.

The second section of Table 4.3 contains a break down of the way edges were added to the network. Only around 40% of edges were created inside the scope of our dataset, the remaining edges predate our snapshots or in rare case not reached by the crawl. Out of all friendships almost a three quarters formed between a pair of existing active avatars. With the remaining 10% forming the same week the avatar was added to the network. So a new (or IA) avatar are forming new edges at roughly 3.75 times the rate among existing avatars.

In all cases the new avatars made more friends in their first week than the IA avatars the formed significant more connections with the existing network than the IA avatars. At the same time the new and IA avatars form friendships among themselves at twice the rate expected from the relative populations which suggests that players will join the game in preexisting groups which are reflected in friendships of their avatars. A network models based on growth through adding smaller graphs in this was explored in [10].

The final section in Table 4.3 contains a breakdown of how edges were removed from the network. As we can see in most cases edges are removed from the graph because a endpoint has been removed. This is normal for social networks since most people hesitate to remove active people from their friends list. For example even in Twitter where people unfollow more freely the number of follows removed is still dwarfed by the number added [18] the same tendency is noted in Cyworld [8]. There are a significant number of

edges removed when both endpoints died simultaneously [2]perhaps indicating that avatar death is weakly contagious. As most edges are never explicitly broken the removal of a edge between a pair of active avatars is a significant event, and off again on again friendships are therefore significant as well.

Broken and unstable relationships

The number of broken and unstable friendships of any given avatar is proportional to its degree, there are no examples of ostracism in the form of mass removal from friend lists. Changing outfits does not usually prompt a mass removal of friendships.

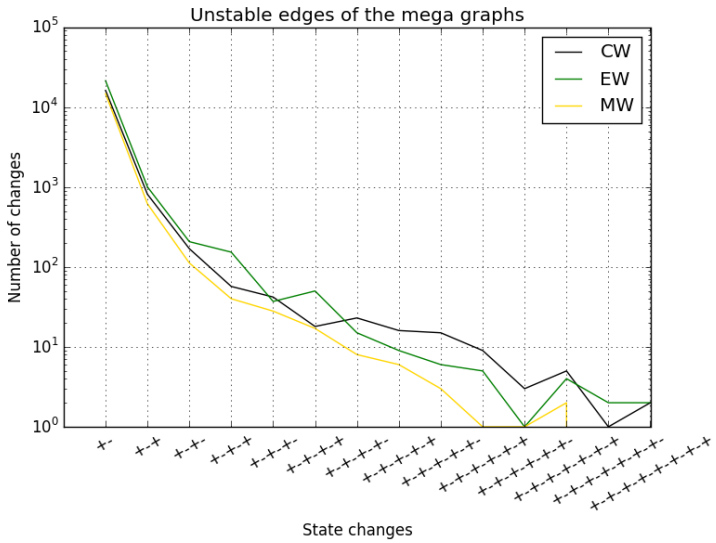


Figure 4.6: On again off again friendships.

[2]potentially this can indicate that one avatar death has cut another off from the main component, however the cutoff avatar would have to be active for 6 snapshots and form no friendships for this to be a problem.

The off again on again friendships for the three servers are shown in Figure 4.6 + indicates a reformation, − a deletion. The graph also includes +− the edges that were simply removed. The combined all of these edges is a little under a tenth the number of edges that are removed as a result of one avatar being abandoned. And highly unstable friendships with more than two off again on again cycles are so rare they can be safely neglected.

When we consider the fact that around 60% of all friendships are over 10 months old, and were never removed by the active endpoint. There are cases where active avatars had links to avatars who had not been online in years, without removing the the link. It is safe to concluded that the removal of nodes is the driving mechanism in this networks evolution rather than the removal of edges.

Avatar death dynamics

One of the important properties of scale free networks is their reliance to random attack [16], due to the power law degree distribution randomly removing vertices is unlikely to break the graph since the vast majority of nodes are low degree. And even if a hub gets hit a only small number of peripheral nodes are likely to get cut off as most are connected in several ways.

In our data low degree peripheral avatars are essentially interchangeable,

huge numbers of low degree mostly new avatars die out each week and are replaced in turn. Around a third of these of the newcomeing peripheral avatars will end up connected to single hub, as we will see in later visualizations like Figure 5.4 and Figure 5.7. Despite this churn the degree of the hubs is usually stable through a combination of new friendships and returning avatars.

When we examined the correlations between avatar atributes we saw that our low degree low clustering perhperal nodes were h

As we saw in the correlations these peripheral avatars tend to be new, rarely spend enough time playing to join an outfit or form more than the first initial friend. In fact the number of avatars who do not even make it to the point of making a friend is enormous [3].

Removal by node attribute

Creation date

First we examine the impact of avatar age on its survival as well as the age of the hubs of the network.

[3]The avatars in our snapshots collectively have 10% more kills then deaths every month, meaning that 10% of all kills are scored on a population of friendless newcomers.

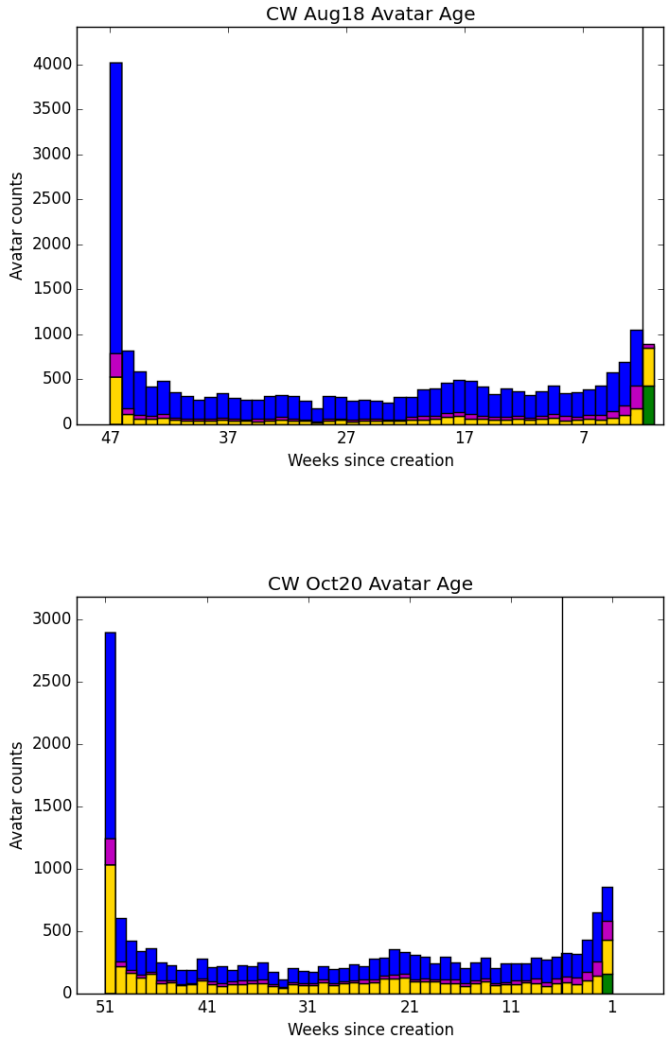


Figure 4.5 breaks the avatars down by the date they were created. The blue bar represents the population of the snapshot as a whole. The yellow bar shows when the avatars who reactivated during the snapshot. The magenta bar indicates when the dead avatars while the green bar represents the new

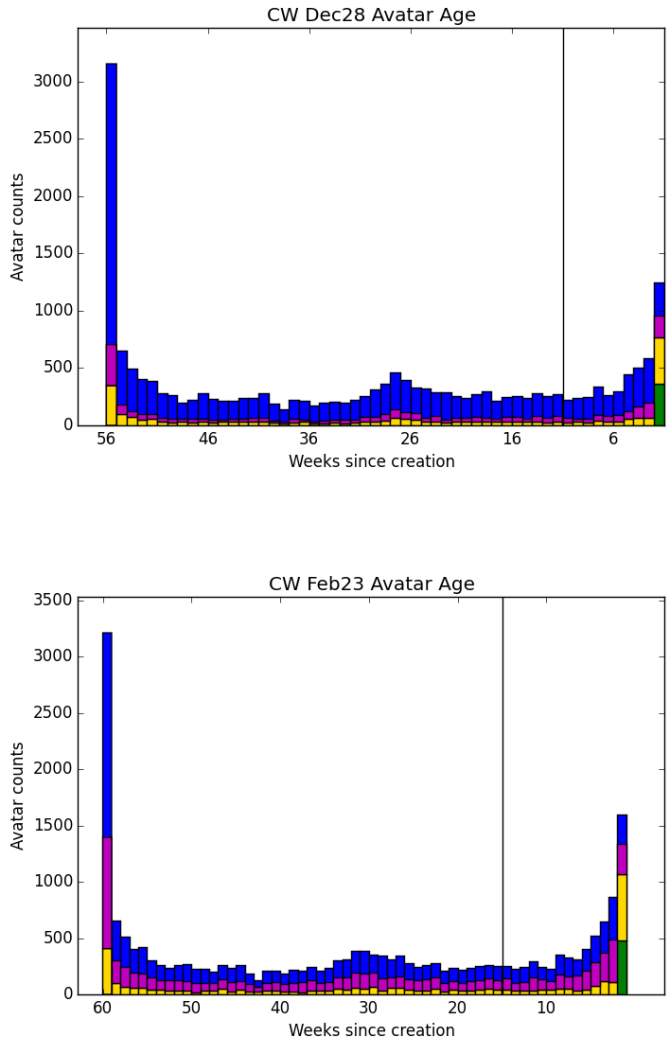
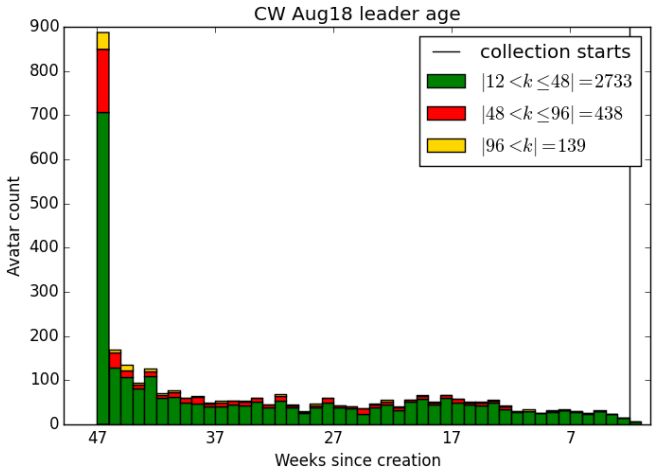
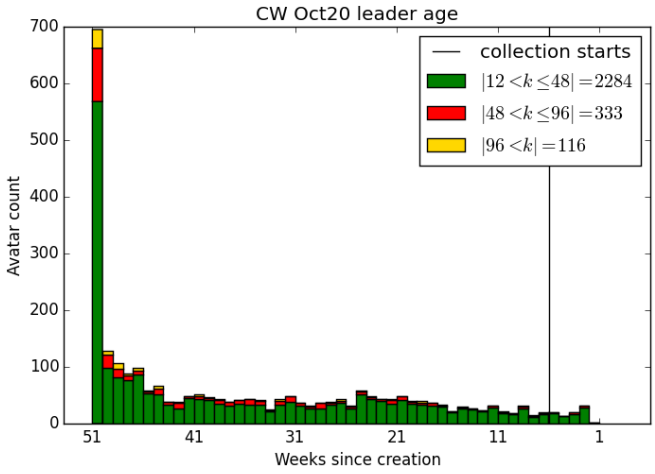


Figure 4.5: Avatars by creation date.

avatars. The black line marks the week we started collecting data. More examples of this are provide in the appendix see Figure 9.2 and Figure 9.1. These all follow similar patterns newer avatars see the highest rate of both return and abandonment, while most of the avatars in any given snapshot were actually created during the launch of the game. The rate at which avatars are created in between these two extremes is surprisingly consistent.





In order to learn more about the hubs of the network we break down the influence of creation date on degree as in Figure 4.4. Avatars with less than 12 friends are not included. The largest hubs mostly date back to launch with the youngest being created over a year before data started being collected.

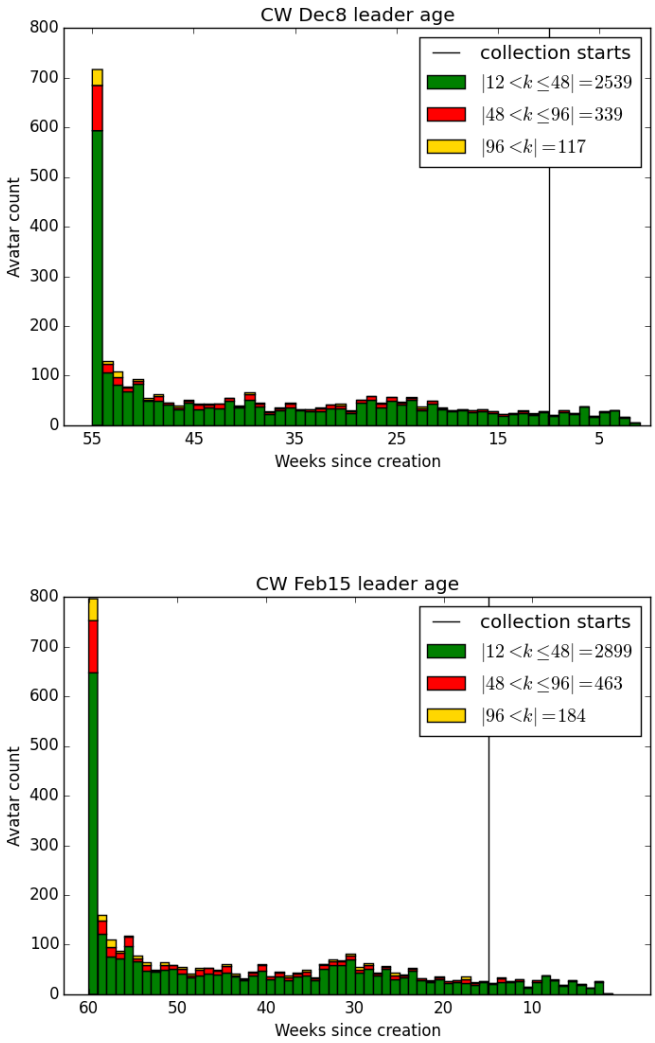
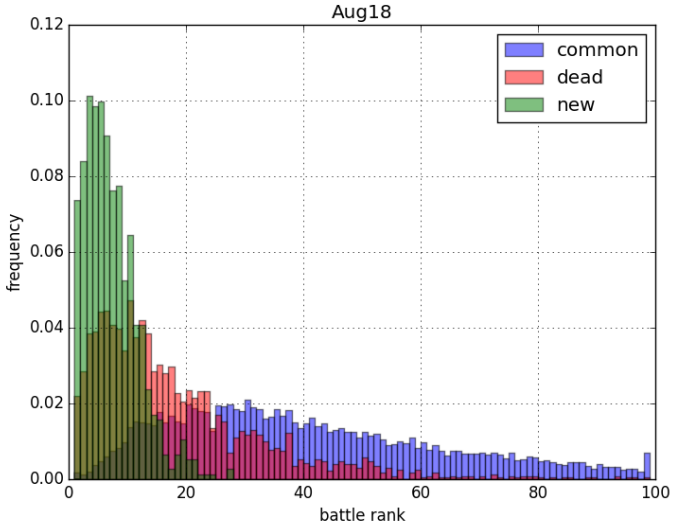


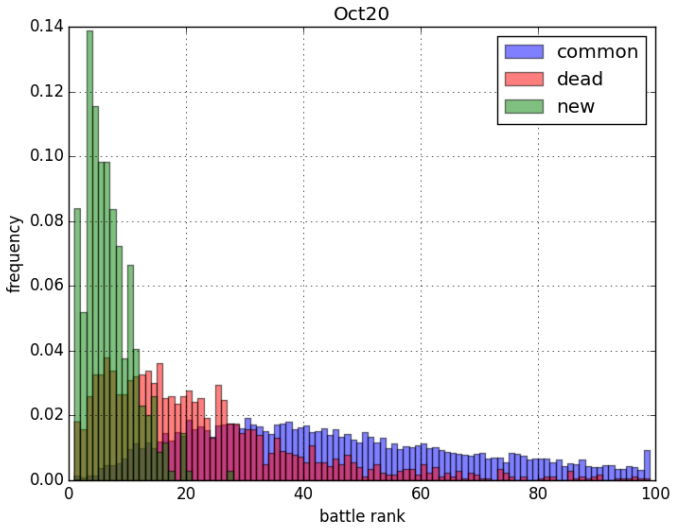
Figure 4.4: High degree avatars by creation date.

The lesser hubs follow the same pattern but less extreme, while the bulk of the none peripheral nodes follow a global tendency after a few months of growth. More examples are provided in the appendix Figure 9.4 and Figure 9.3. So the higher degree avatars are also the oldest, and there is little upward mobility among the new nodes after a point.

By battle rank

Earlier we noted the strong correlation between battle rank, time played and skill statistics. This section addresses the impact of these on survival. Only examples for battle rank are included since the others give the same results.


Aug18
common
dead
new
frequency
battle rank
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0
20
40
60
80
100


Oct20
common
dead
new
frequency
battle rank
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
0
20
40
60
80
100

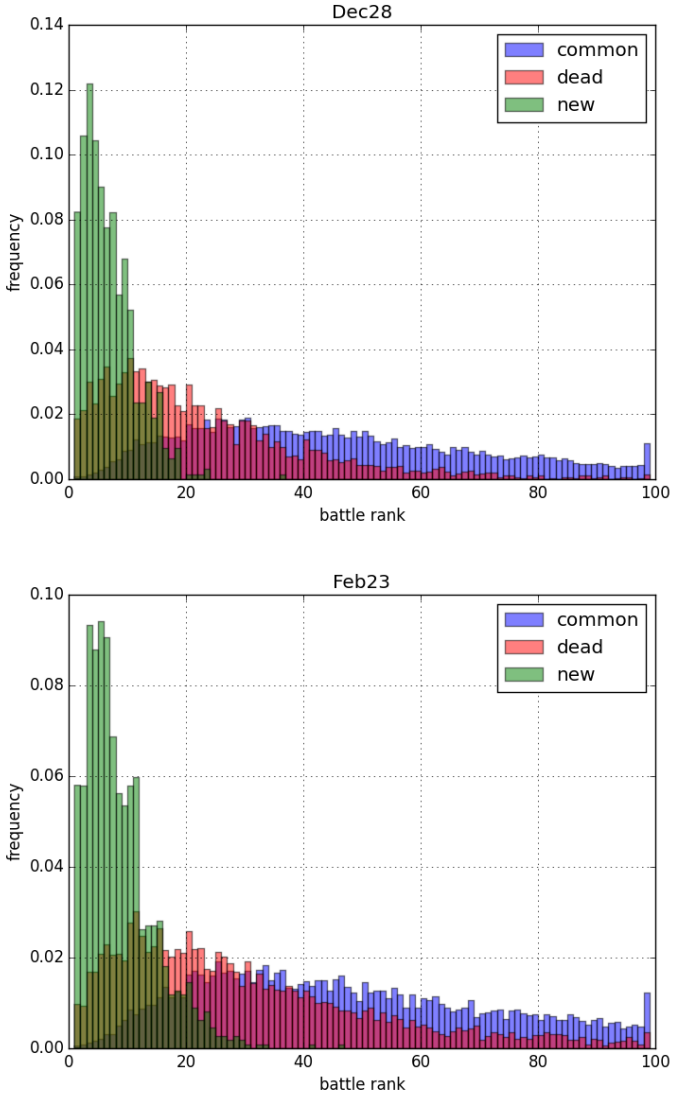


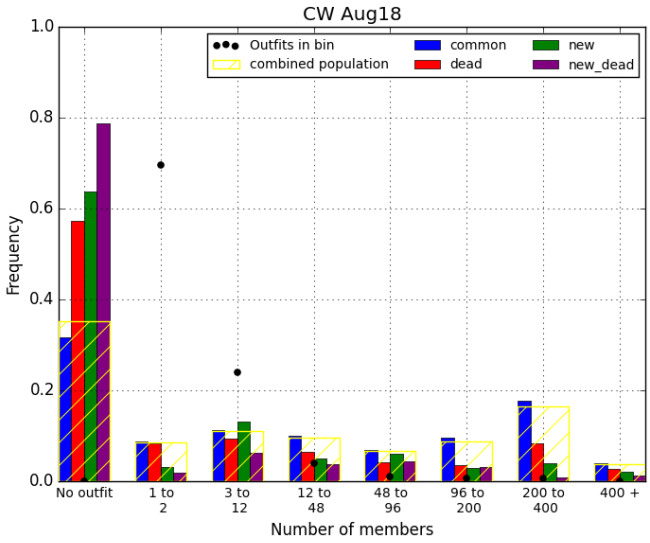
Figure 4.3: The normalized battle rank distribution.

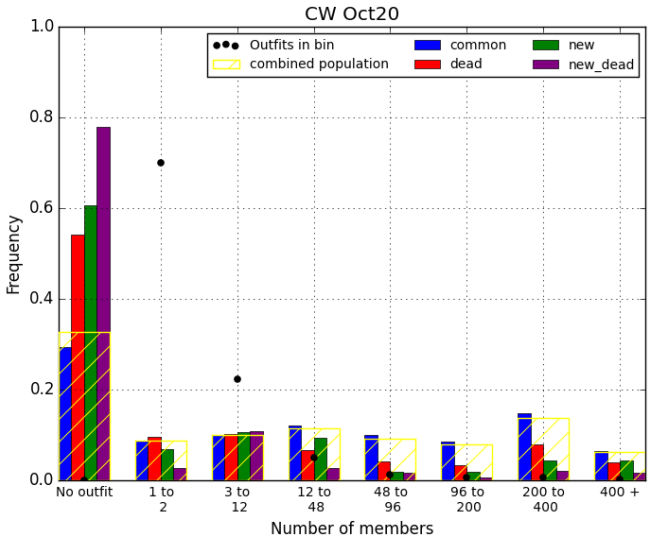
In the Figure 4.3 we compare the battle rank of the new dead and IA avatars, the values are again normalized for the sake of comparison. Examples from the other servers are available in the Appendix 9.7. New avatars are

mostly concentrated below rank 20, which is consistent with the amount of progression possible in a single week. The number of dead avatars shrinks relative to the overall population as battle rank increases. The position of the IA avatars can be inferred from the overlap of the new and dead avatars. As expected the higher the battle rank the less likely the avatar is to die, the same trends apply to time played and kill to death ratio.

By outfit size

Finally we examine the impact of outfit membership on survival rates. The avatars were broken down by the size of outfit in active avatars, and by state resulting in Figure 4.2.


CW Aug18
Outfits in bin
combined population
common
dead
new
new_dead
Frequency
1.0
0.8
0.6
0.4
0.2
0.0
No outfit
1 to 2
3 to 12
12 to 48
48 to 96
96 to 200
200 to 400
400 +
Number of members


CW Oct20
Outfits in bin
combined population
common
dead
new
new_dead
Frequency
1.0
0.8
0.6
0.4
0.2
0.0
No outfit
1 to 2
3 to 12
12 to 48
48 to 96
96 to 200
200 to 400
400 +
Number of members

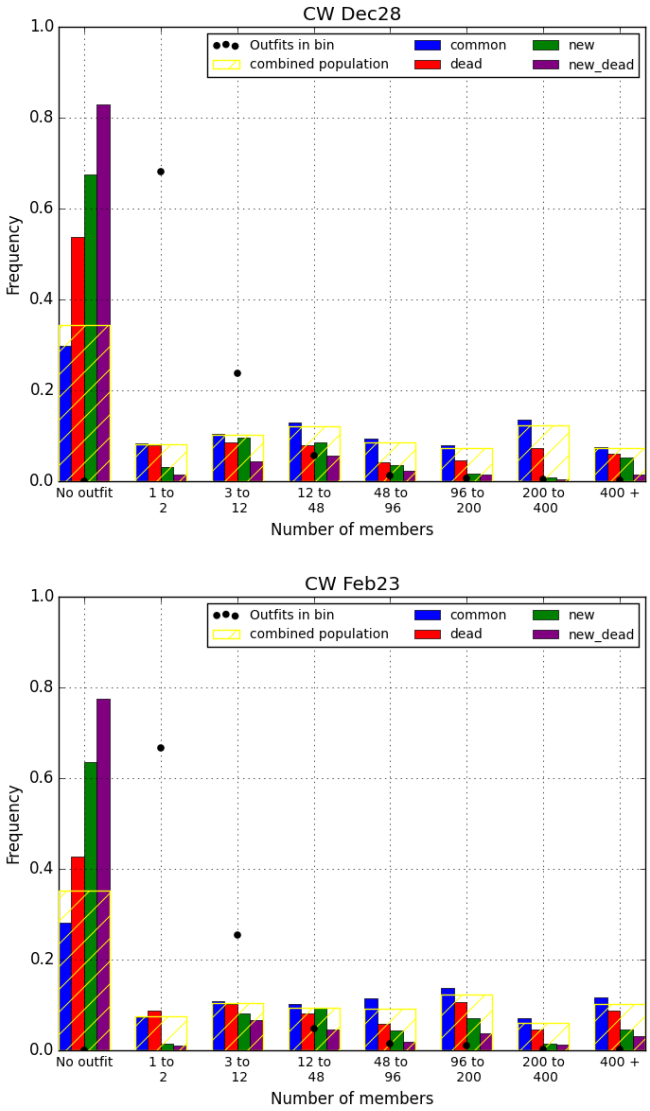


Figure 4.2: Avatar state by size of outfit.

The black points show the number of outfits of that size in the range, the vast majority of outfits have a 1 or two members consisting of either brand new outfits or the last remnant of former ones. The larger outfits have the

population spread evenly between them, the break down of population by outfit size agrees with the literature [7, 23]. And finally a third of the overall population and the majority of new and IA avatars do not in a outfit.

Being in a tiny outfit alone or with one other members should be as bad as being independent, despite this the proportion of dead avatars in such groups is still lower. In general the 12 - 48 and 48 - 96 member outfits tend to be best for the survival of their members. While the small and extremely large outfits recruit the greater proportion of the new avatars. As the outfits gradually shrink, with more remnant 1 or 2 member outfits holding more of the population.

Summary

Clearly the main controller of friendship removal in the network is the removal of avatars themselves as edge removal is rare. However, the majority of the network evoloution models that exist for social networks are based based around edge rewiring mechanisms instead [29]. Comparing theses models to those that better match the real world mechanism might be a good future project.

In conclusion snapshots indicate that high degree high clustering improves a avatars lifespan. Membership in an outfit is good for retention assuming it is of respectable size. But, in general past commitment as measured by time

played and battle rank predict that the avatar will continue to be played in the future. The difficulty comes from identifying which way the causation runs. Are players who like the game naturally acquiring a stronger social position by exposure or are players bound in by a stronger social network compelled to continue playing. Or some mixture of the two?

Chapter 5

Server Merger

The initial motivation for gathering the data from PlanetSide 2 was to record an upcoming server merger. As far as we know there are no existing studies of this kind of event in the literature, especially none on this scale. There are few direct real world analogs for the merger of two servers, though the merger of two tribes or corporations would be a good example of existing social networks merging. So we wanted to see what happens, the rest of the section examines the mixing process of two formerly isolated groups now abruptly thrust together.

First we give some background and a short explanation of what a server merger is in the context of a MMOG, then we examined the mixing of the two populations graphically and analytically.

The server merger

First we will go over what a server merge is and what it means for a MMOG. With a few notable exceptions MMOG like all multiplayer video games begin to lose players a few weeks after launch as they move on to other games. However, unlike single player and normal multiplayer games, massively multiplayer games require a very large number of simultaneous players to function as intended. A server merger is a common response to dwindling populations.

This is especially true for PlanetSide 2 as a purely player versus player game, as without other players there is nothing to do. In this case the avatars from both servers are placed onto the same renamed server. [1]

The merger data

In early spring of 2014 SOE announced they would merge the two US west coast servers, Mattherson and Waterson. The merger took place in July 2014. We wrote the first version of our friendslist crawler described in chapter 2 and 8. Capturing snapshots of both servers on the 18th of May and the 15th and 23rd of June. Unfortunately, this version of the crawler only collected the following avatar attributes: battle rank, outfit and avatar name. The first

[1] They actually held a two hour 400 vs 400 player competition to see which name would be kept. However it ended in a draw so they compromised and named it Emerald

five snapshots of the new emerald server (June 30, July 14, July 22, July 29 and Aug 4) also used the old code.

We expand the data range of data captured in the week of August 11th but the most critical information on the merger does not have all the information described in chapter 2. Therefore, for the rest of this chapter, we must use the entire 44 days of data and only look at the avatar attributes of battle rank outfit and the network itself.

Graphical evolution

Reading the graphs

The images in this section were created using Gephi [4] with the force atlas 2 layout using default settings for a network of its size. Each avatar is assigned a colour according to Table 5.1.

Origin	Faction		
	NC	TR	VS
Waterson			
Mattherson			
Neither			

Table 5.1: Merger: Colour key

The size of each avatar scales linearly with its degree, with one exception the highest degree avatar is scaled down to be no more than twice the size of second highest degree avatar. This was necessary to make the other hubs to be visible. The edges are left grey for clarity. To get a clear image the screen shot needs to be massive. We found a 4024 by 4024 image to be sufficient to get a useful level of detail.

Each visualization is accompanied by a pie chart showing the population by server of origin, and bar graphs showing the number of friendships between avatars of different origins. The left bar graph counts edges whose endpoints are members of the same faction, while the right bar graph the number of edges connecting members of different factions together. The pie charts and bar graphs use the same colour scheme as the avatars in the visualizations.

The servers before the merger

Each of the three massive lobes correspond to one of the three factions, note the small number of avatars from different factions that connect to only enemies. Each of the factions will have at have a few massive hubs in its core together, as well as connecting to a large number of degree 1 or 2 avatars that would otherwise be isolated. Each server usually has a single super massive hub with a degree 2 to 6 times that of the next highest degree avatar.

These visualizations in Figure 5.2 and Figure 5.4 are typical examples of a snapshot in a normal server. For comparison to normal social networks there are two example networks from [15] under the same visualization algorithm in the appendix. Figure 9.8 the collaboration network of astrophysics papers, Figure9.9 is the network of who trust whom in Epinions. When looked at individually the faction lobes are similar to these typical social networks.

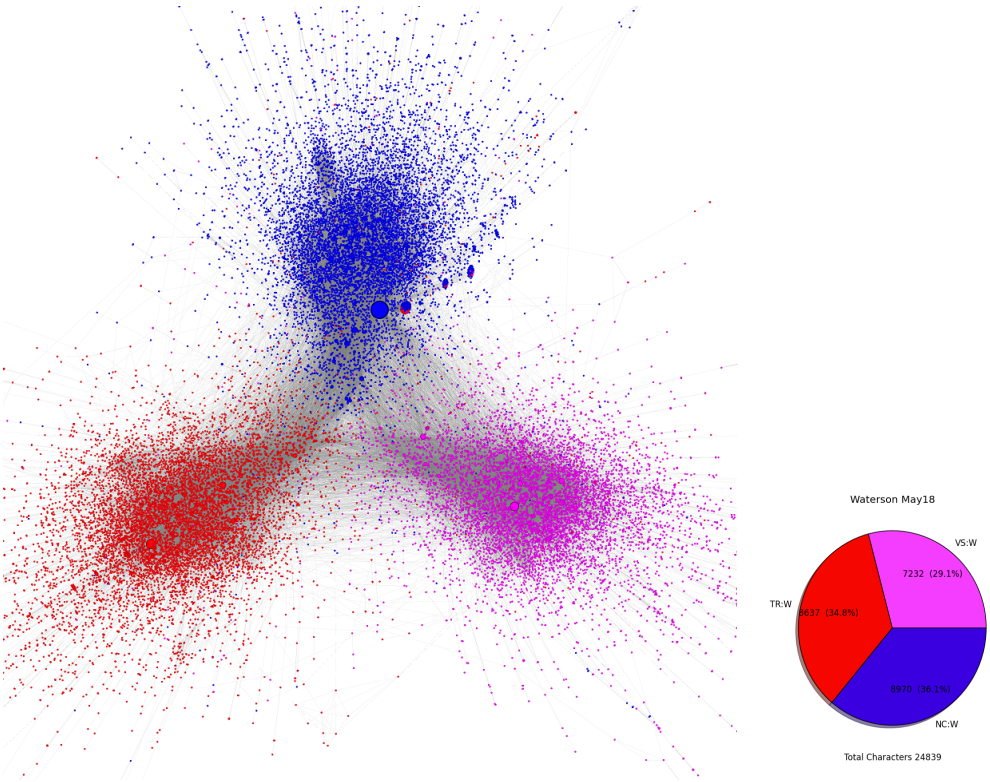


Figure 5.1: Merger: Waterson May 18

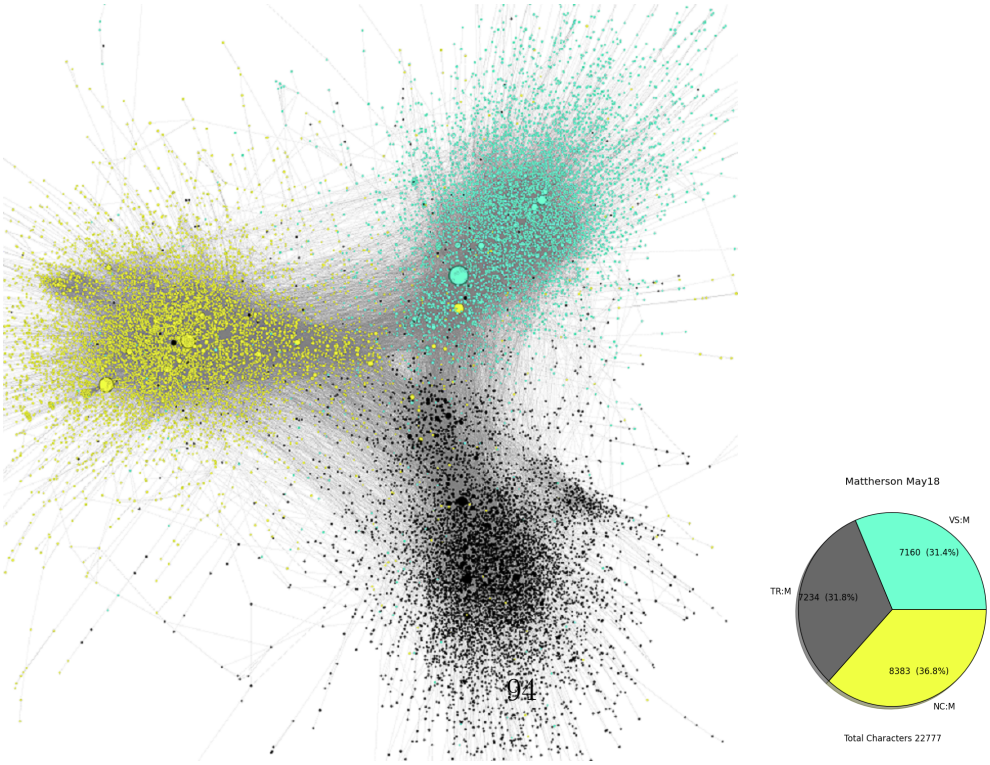


Figure 5.2: Merger: Mattherson May 18

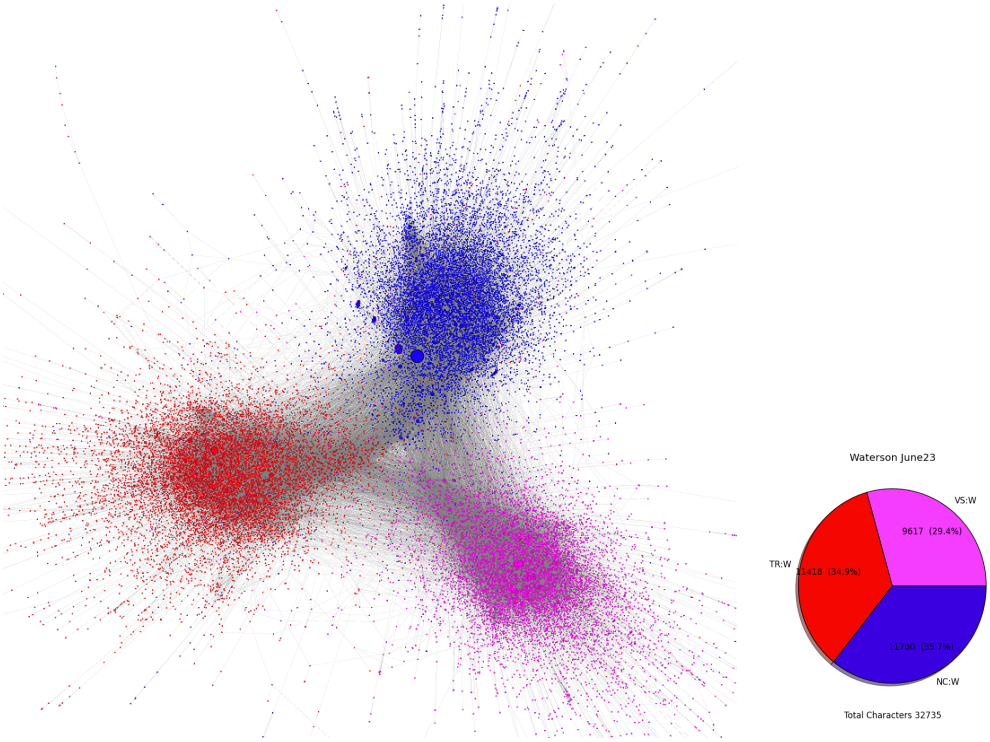


Figure 5.3: Merger: Waterson June 23

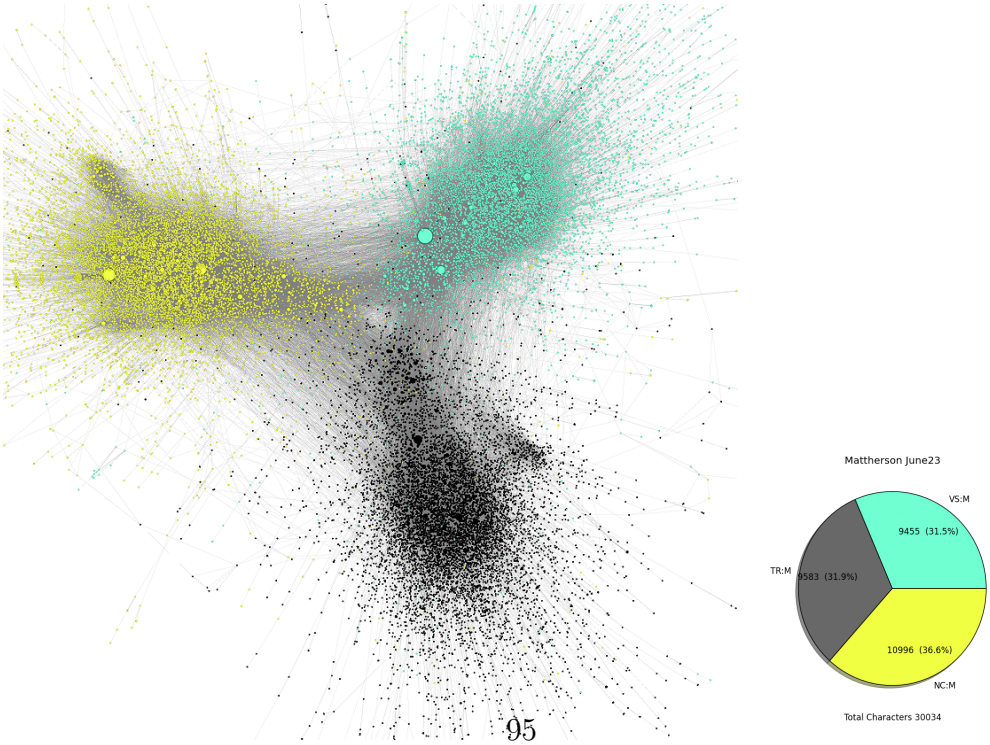
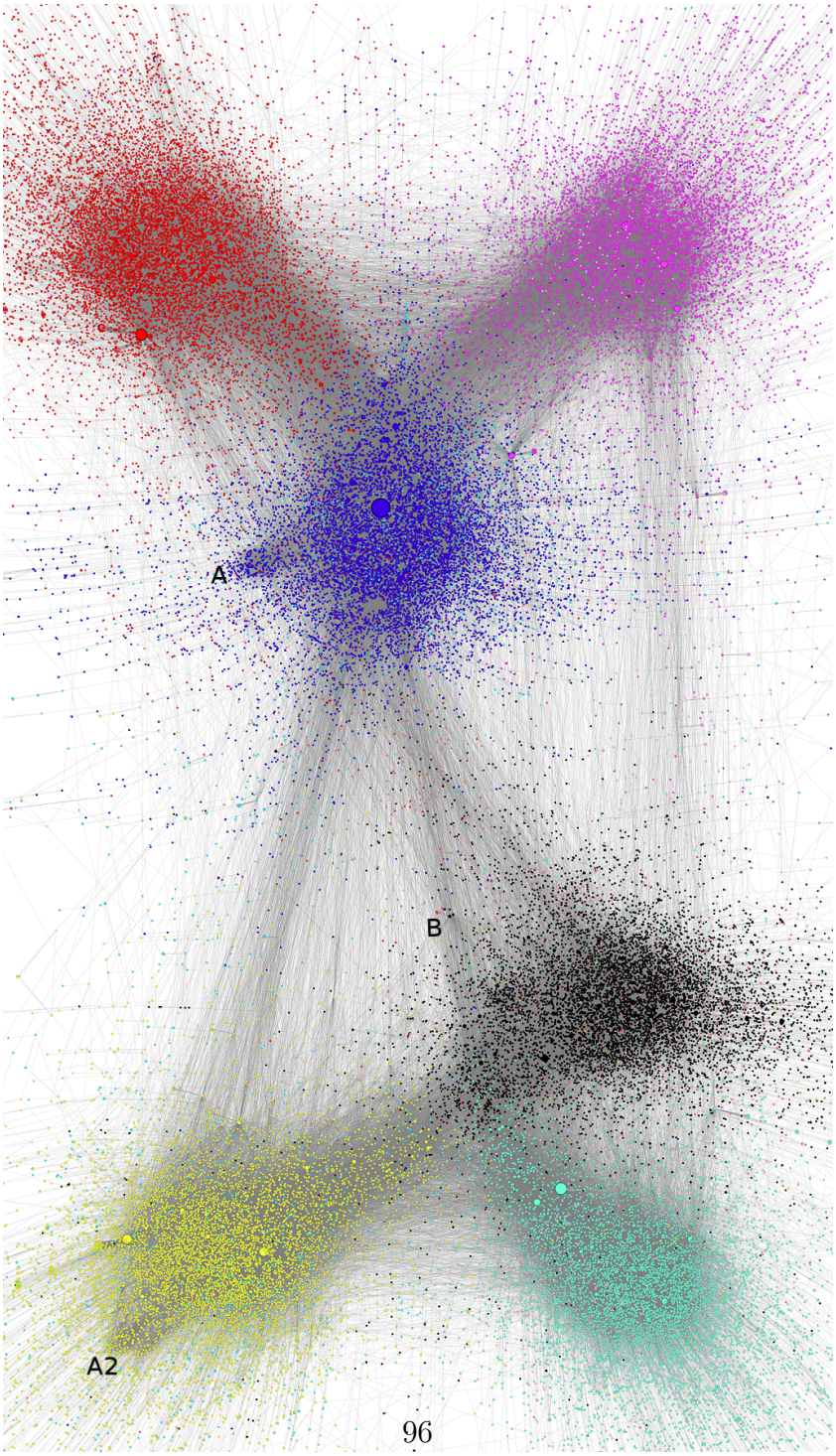


Figure 5.4: Merger: Mattherson June 23

The merger



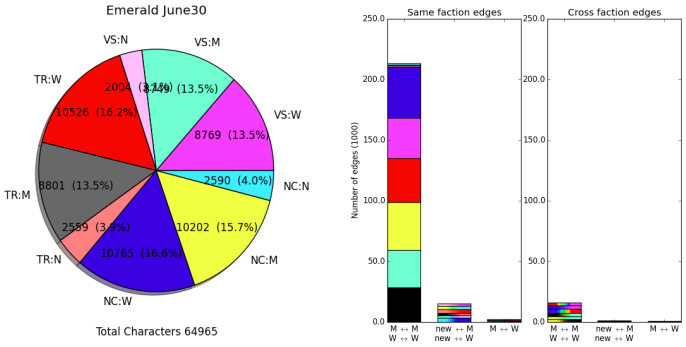


Figure 5.3: Merger: Emerald June 30th

June 30 is shown in Figure 5.3. This is the first snapshot of the newly formed Emerald server, already 11% of the population is from neither of the original servers. When we look at the edge break down we can see the initial $M \leftrightarrow W$ edges between endpoints who are members of the same faction are outweighed by the cross faction edges of the original server however the number of edges connecting the servers of both origins to new avatars of the same faction already equals the number of cross faction edges between the original servers. As we will see later in Figure 5.17 the majority of the early cross origin edges are formed between hubs and the peripheral avatars. However, at this point none of the major hubs have formed cross origin edge with their counter parts.

The letter A marks an outfit whose members are unusually strongly connected internally while being relatively insular, preventing it from being pulled into the core of the network. A2 marks another case consisting of two such outfits that overlap heavily with each other but not the rest of the faction.

Outfits like these are fairly rare, while many outfits have strong connection between members, they often have a hub that dominates this structure by pulling it into the core of the graph, or most members will be just as strongly connected to outsiders or some mixture of the two will prevent them from being easily seen in the visualizations.

The B marks an avatar of a YouTube celebrity in the broader game community, which is immediately pulled into a position between the two servers, appearing as a broker [16, 24].

Figure 5.4 shows the July 14 snapshot by this point 28.8% of the population are from neither of the originals. Over the course of the month of July the factions begin to slowly bind themselves together, by the time of July14 the edges bridging the gap between the Mattherson and Waterson avatars has begun to outnumber the connections between them and their old opponents.

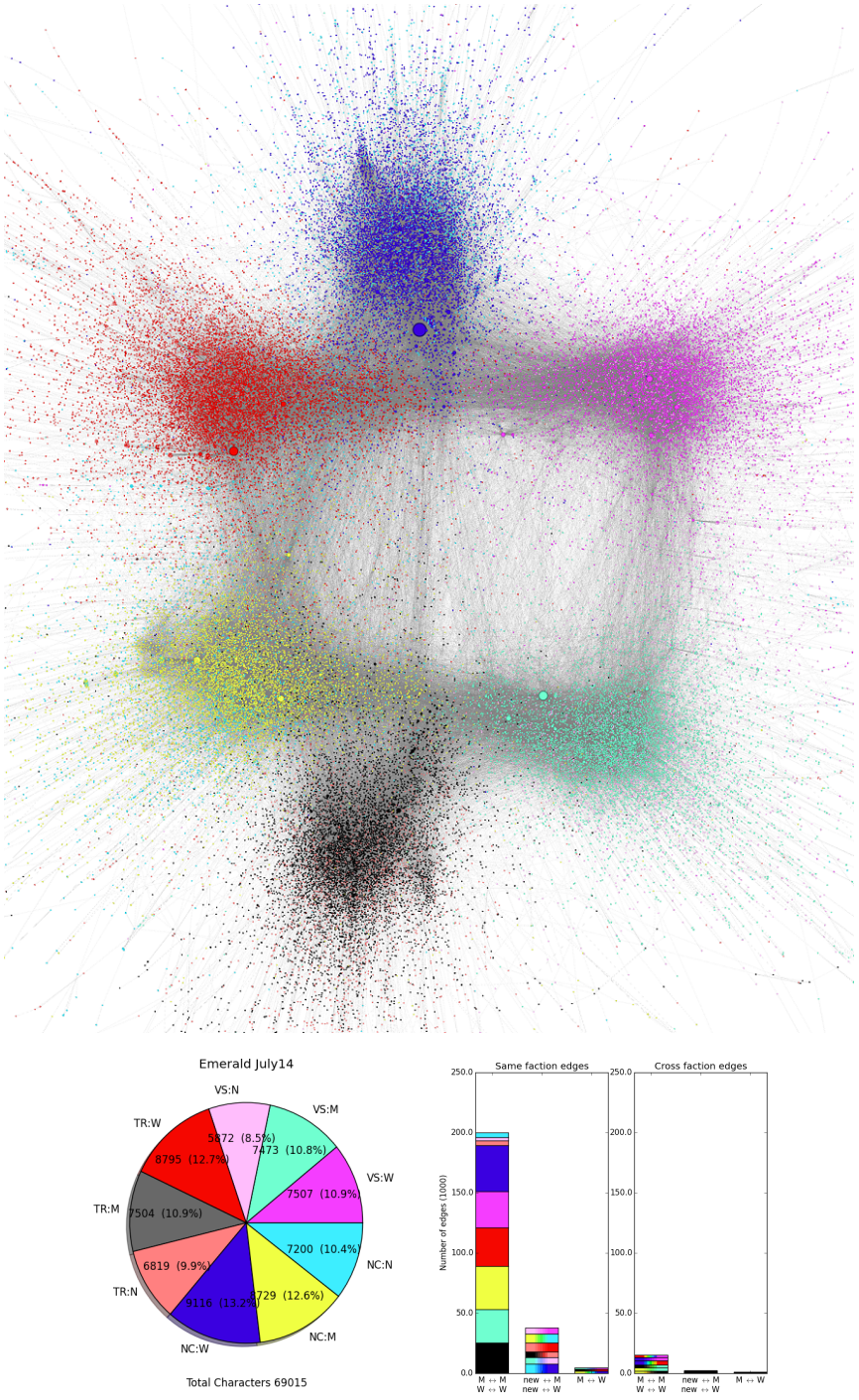


Figure 5.4: Merger: Emerald July 14th

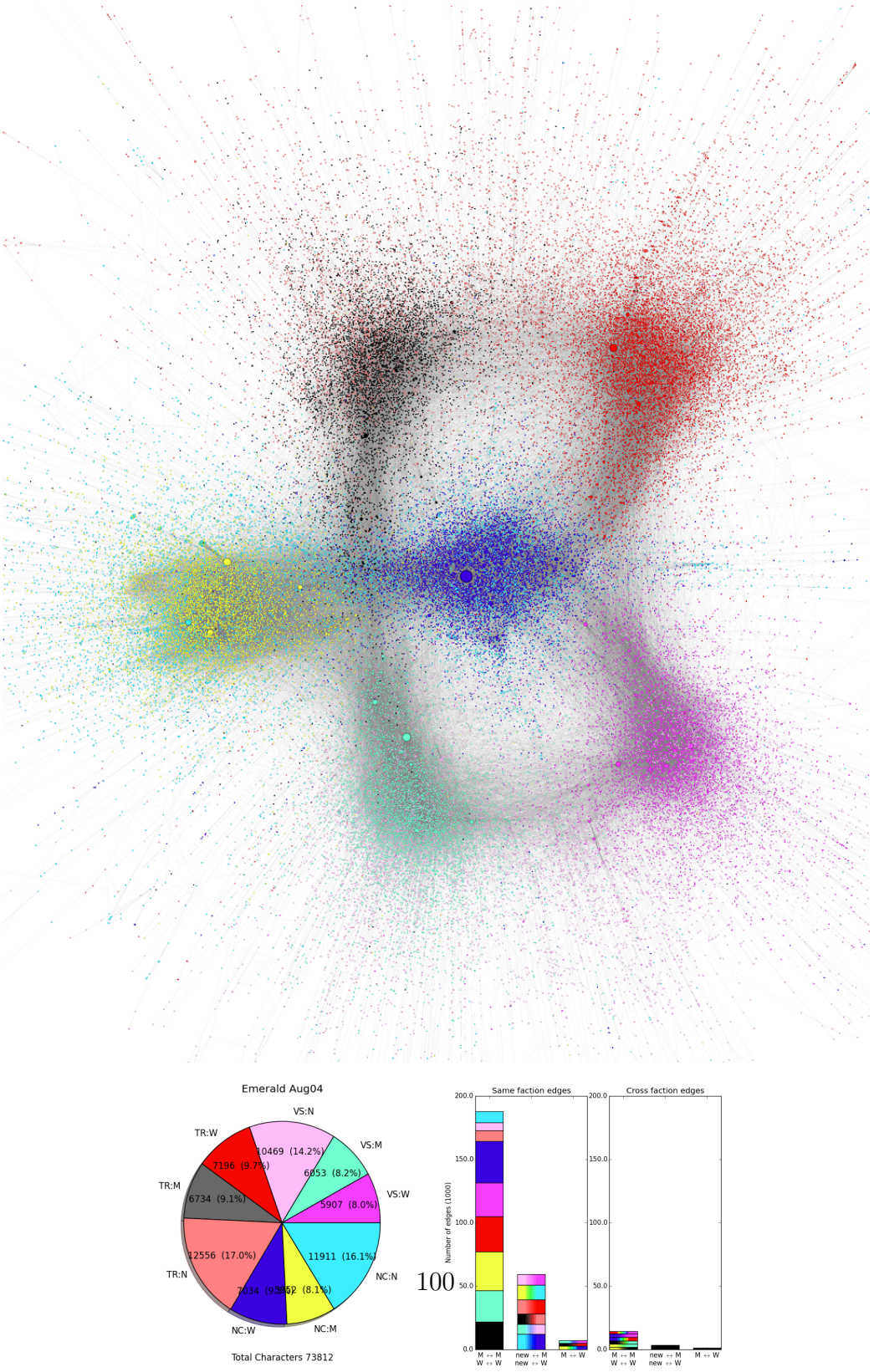
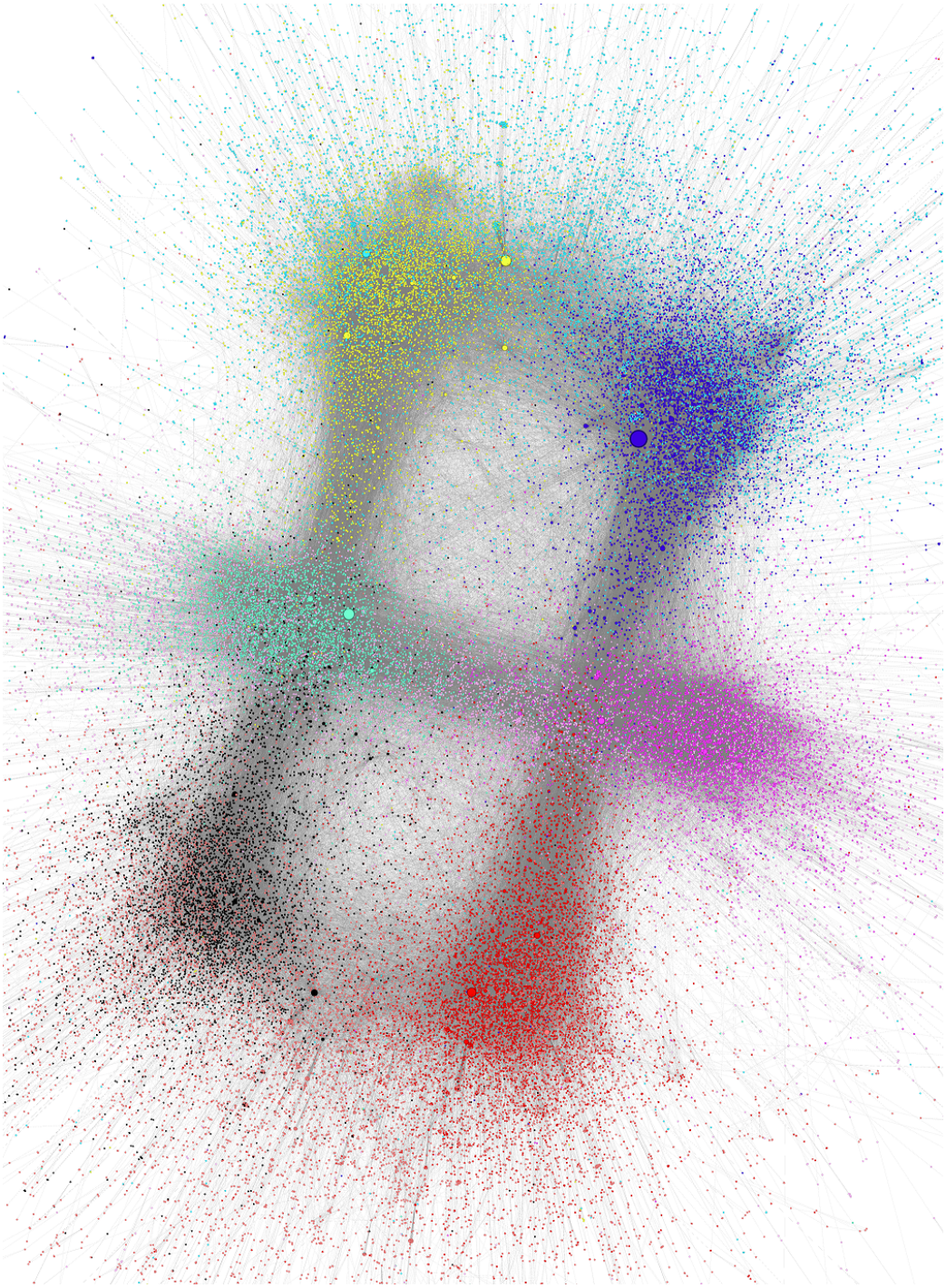
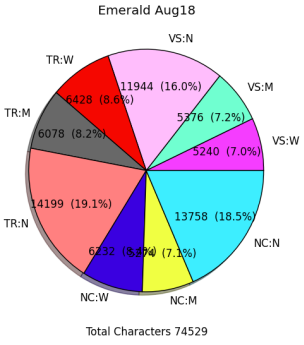
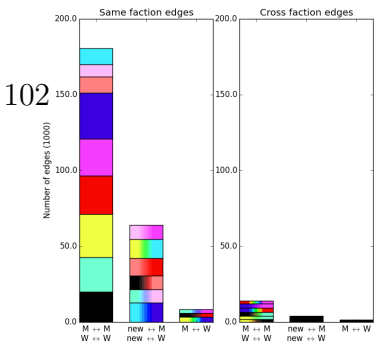


Figure 5.5: Merger: Emerald Aug 4th

By the fourth of August as seen in Figure 5.5 the number of newcomers and edges connecting has nearly doubled, while the number of cross origin edges has doubled it is still not enough to overcome the combined bonds ties to their original enemies. Little has changed since the last snapshot other than the graphs drawing closer.




Emerald Aug18
VS:N
TR:W
VS:M
TR:M
VS:W
TR:N
NC:N
NC:W
NC:M
11944 (16.0%)
6428 (8.6%)
5376 (7.2%)
6078 (8.2%)
5240 (7.0%)
14199 (19.1%)
13758 (18.5%)
6232 (8.4%)
5297 (7.1%)
Total Characters 74529


Same faction edges
Cross faction edges
Number of edges (1000)
200.0
150.0
100.0
50.0
0.0
M ↔ M
W ↔ W
new ↔ M
new ↔ W
M ↔ W

Finally on the 18th of August as seen in Figure 5.6 is the first snapshot where the number of cross origin edges connecting avatars in the same faction from the two original servers outweighs the combined number of edges binding them to their original enemies. This results in a collapse back to a three lobed shape.

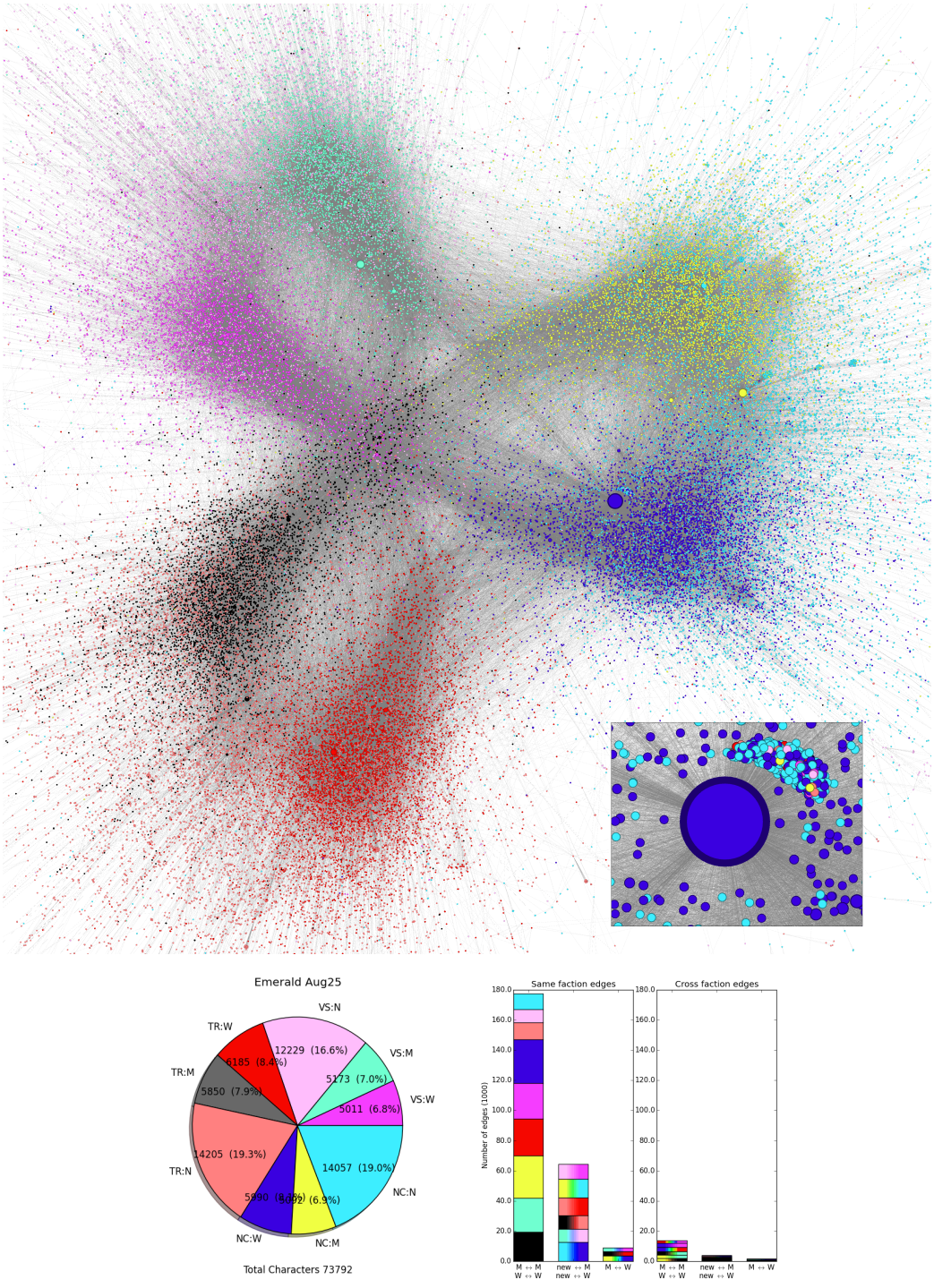


Figure 5.7: Merger: Emerald Aug 24th.

One snapshot later and we have a structure similar to what we started with as we can see in Figure 5.7. We also expanded the area around the highest degree avatar Klypto who has over 5100 friends in the current snapshot, of those over 450 have no connections besides Kylpto. This is actually almost 4000 more friends than the next highest. And has so many cross faction connections to the Waterson TR it actually deforms the shape of the newly combined TR lobe. While each server has at least one supermassive hub Klypto held the record by a factor of three at his height.

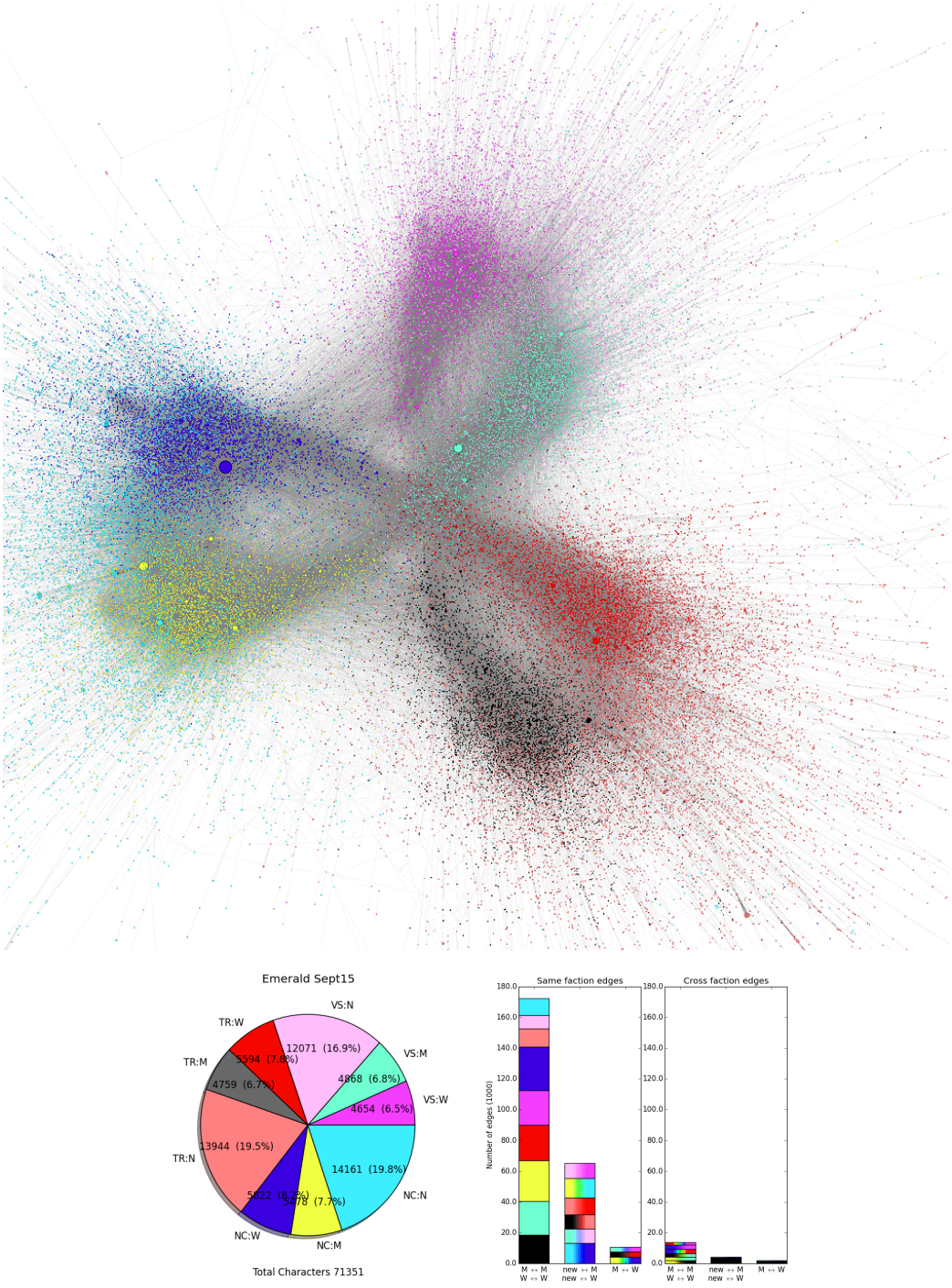


Figure 5.8: Merger: Emerald Sept 15

In Figure 5.8, we see that the original peripheral avatars have been mostly replaced with newcomers. And as a result avatars from the original servers are now the minority. However, the two original cores persist in each faction.

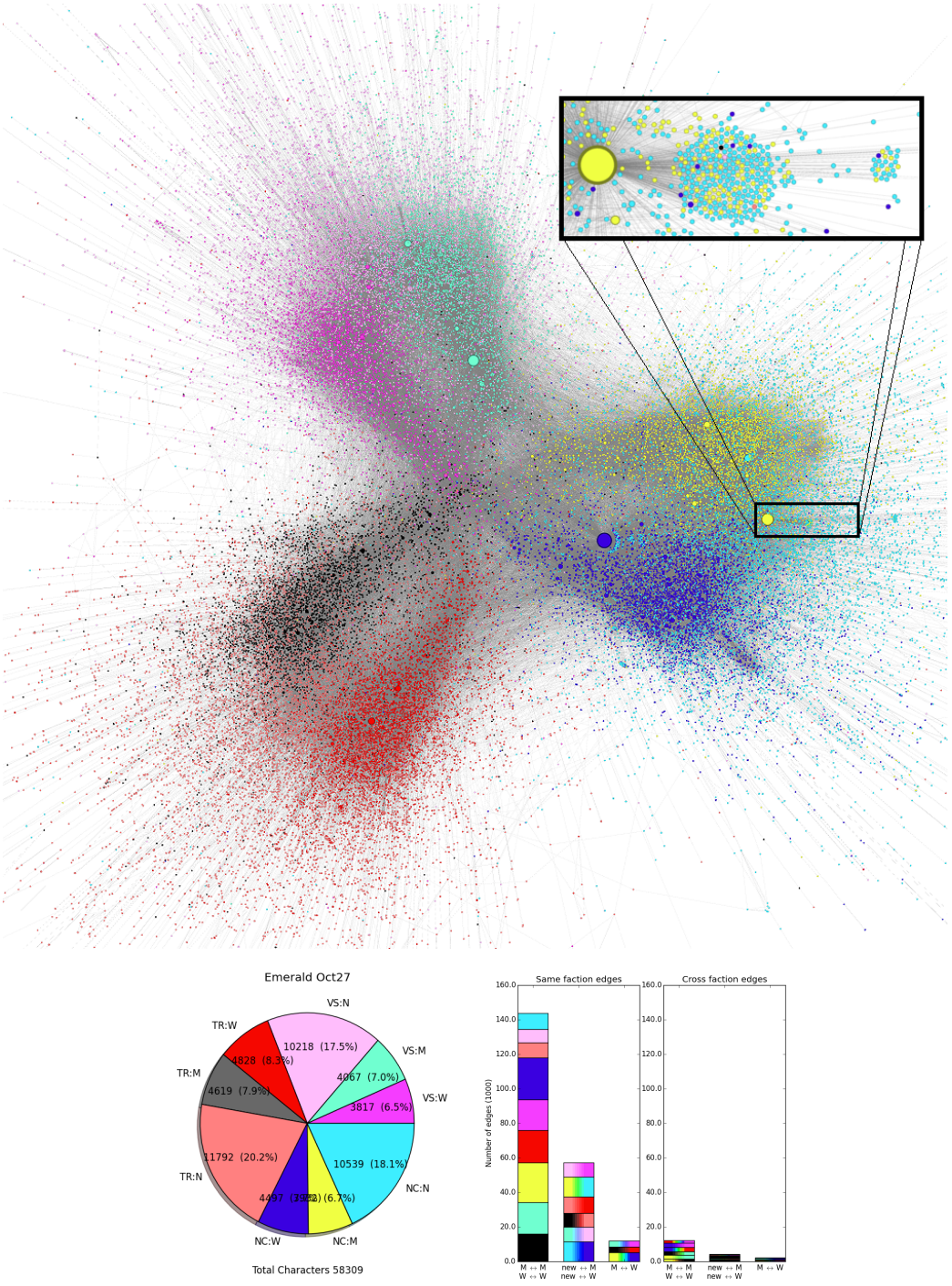


Figure 5.9: Merger: Emerald Oct 27th.

In Figure 5.9 we have zoomed the avatars who are connected to the second highest degree avatar. The avatars in the center have no neighbors other than the hub, those in the right cluster are all connected to exactly one other friend in the same cluster. Finally those on the right near the hub are connected among each other and the network as a whole. This is a common structure found near hubs of almost any size and the main cause of the large diameter we see in Chapter 2.

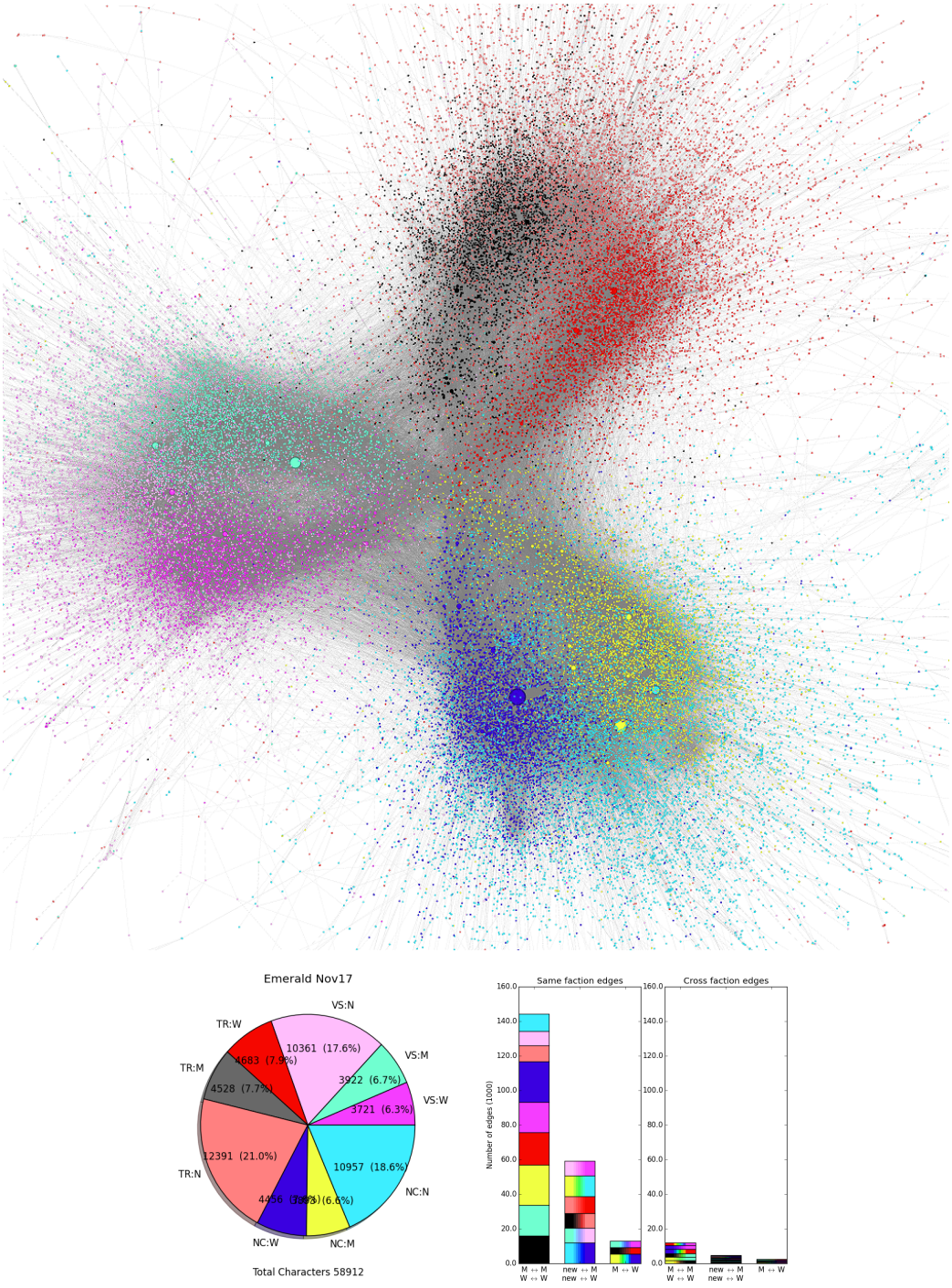


Figure 5.10: Merger: Emerald Nov 17th.

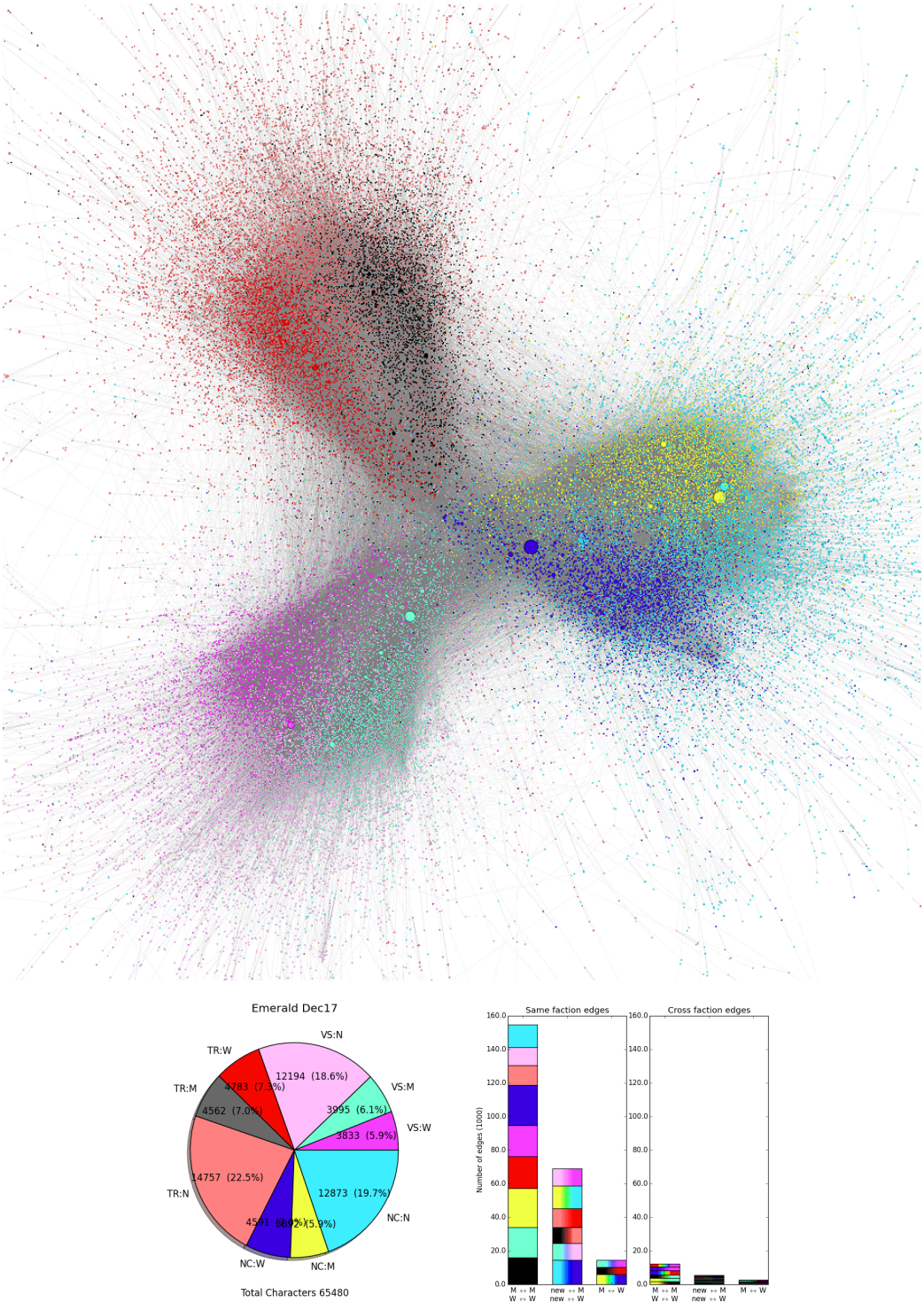


Figure 5.11: Merged Emerald Dec 17th.

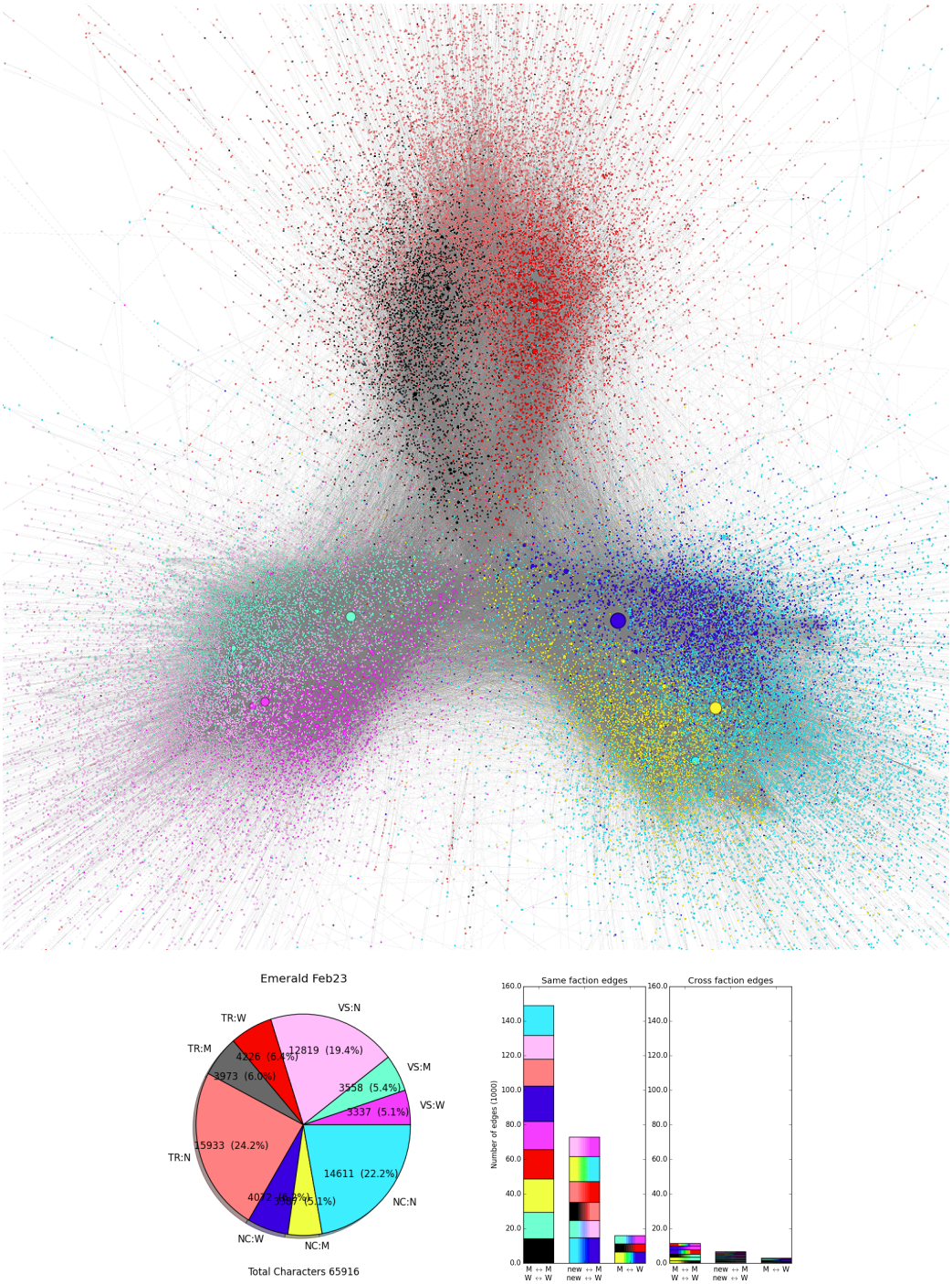


Figure 5.12: Merger: Emerald Feb 23rd.

See Figures 5.11 5.12 for the last two months. It finally becomes difficult to visually tell Waterson or Mattherson apart without the aid of the colours. However, when we look at the assortativity in Figure 5.14, the tendencies to only mix among themselves is clearly still going strong.

In the final snapshot in Figure 5.12 the number of edges between the Mattherson and Waterson avatars still do not outnumber those connecting them to themselves. This is despite the number of avatars remaining from the original servers dwindling down to about 6% of the population. Even more interestingly, the majority of edges are incident to the original avatars.

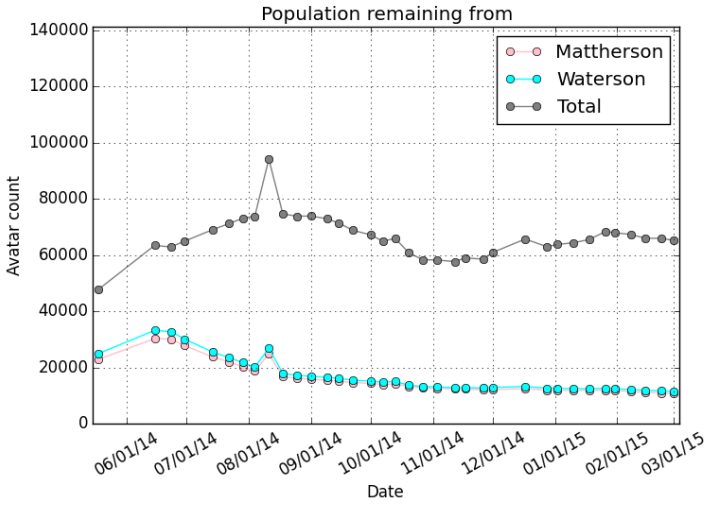


Figure 5.13: The population of avatars by origin.

The lines in Figure 5.14 are broken down as follows: The NC, TR and VS lines show the assortativity within each respective faction, the overall is for of all 9 origins, and original is the assortativity when only avatars from

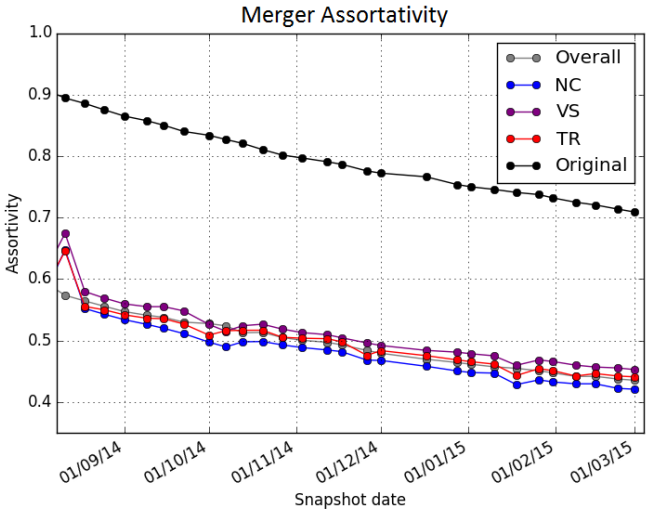


Figure 5.14: The assortativity by origin.

the original two servers are considered. This can also be seen in the average degree of the avatars from the original servers. As shown in Figure 5.15, the average degree of the avatars from Mattherson and Watterson actually grows as the lesser nodes are gradually striped away and replaced with newcomers.

Degree differences

In this section we look at the difference in the degree of endpoints of the edges between various origins. The degree differences between the new avatars the first and last snapshots is shown in Figure 5.16. On the 30th of June the difference is small since these new avatars haven't had time to become high degree. By March 32nd a number of hubs have arisen and are

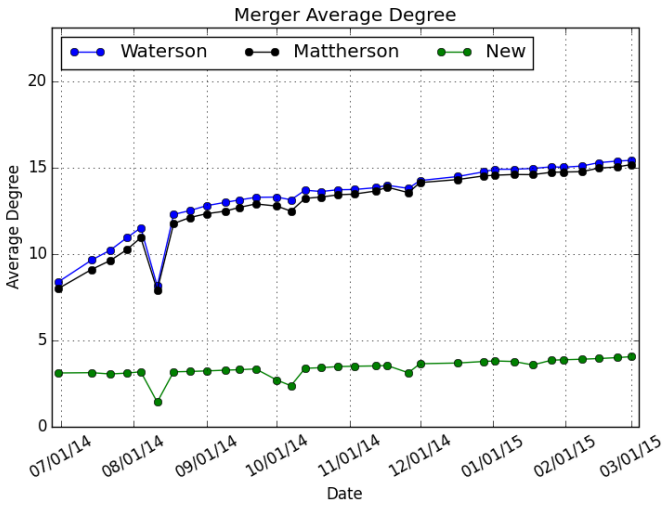


Figure 5.15: Average degree by origin.

pushing the distribution into line with the values of the original avatars.

Figure 5.18 shows the degree difference of edges interconnecting avatars from the original six factions during the June 30th and March 1st snapshots respectively. The values for edges connecting the newcomers to the existing network are shown in Figure 5.17.

As we can see in Figure 5.17 most of the friendships that connect Matther-son avatars to Watterson avatars of the same faction are of very different degree, confirming our thoughts about the connections among the original servers.

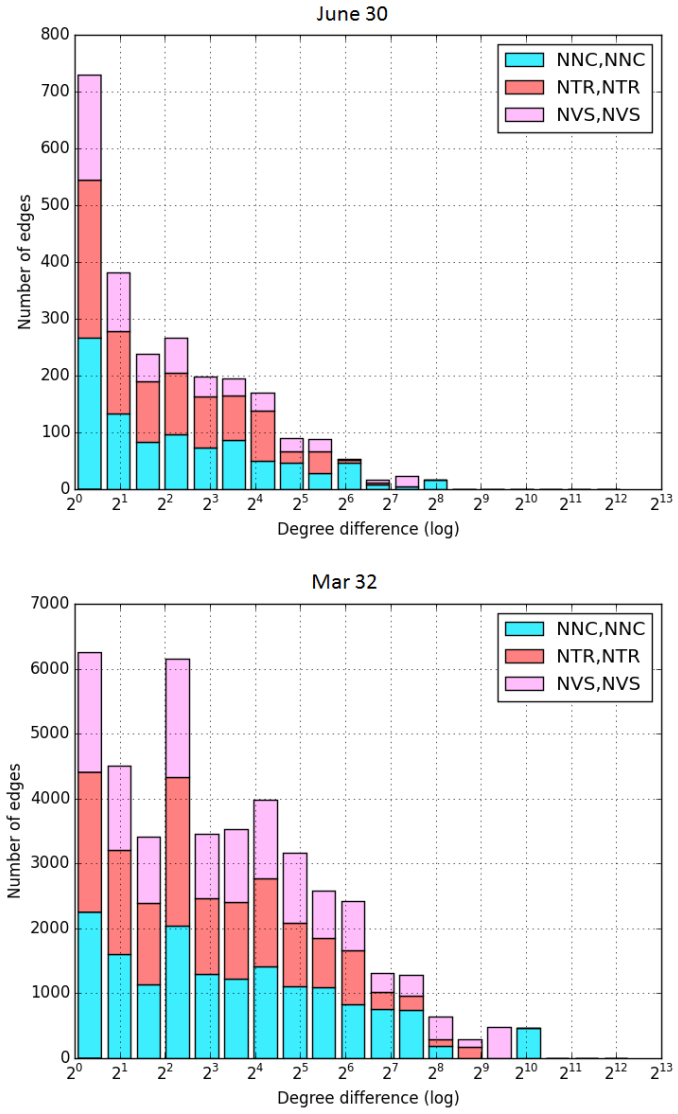


Figure 5.16: Degree differences of new avatars.

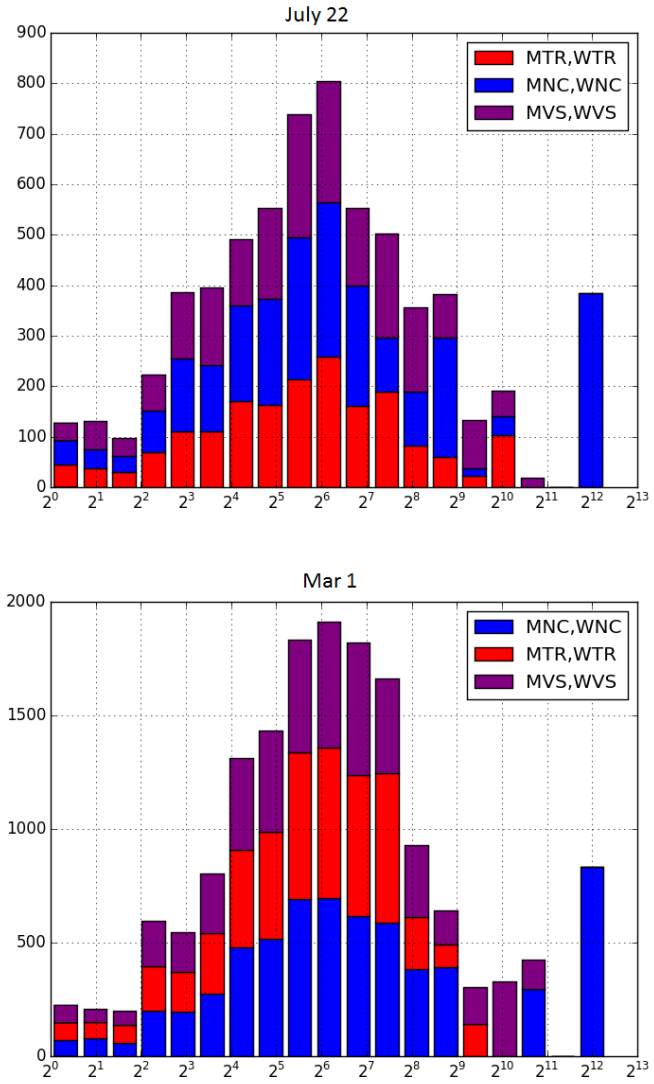


Figure 5.17: Degree difference of mattherson to waterson edges.

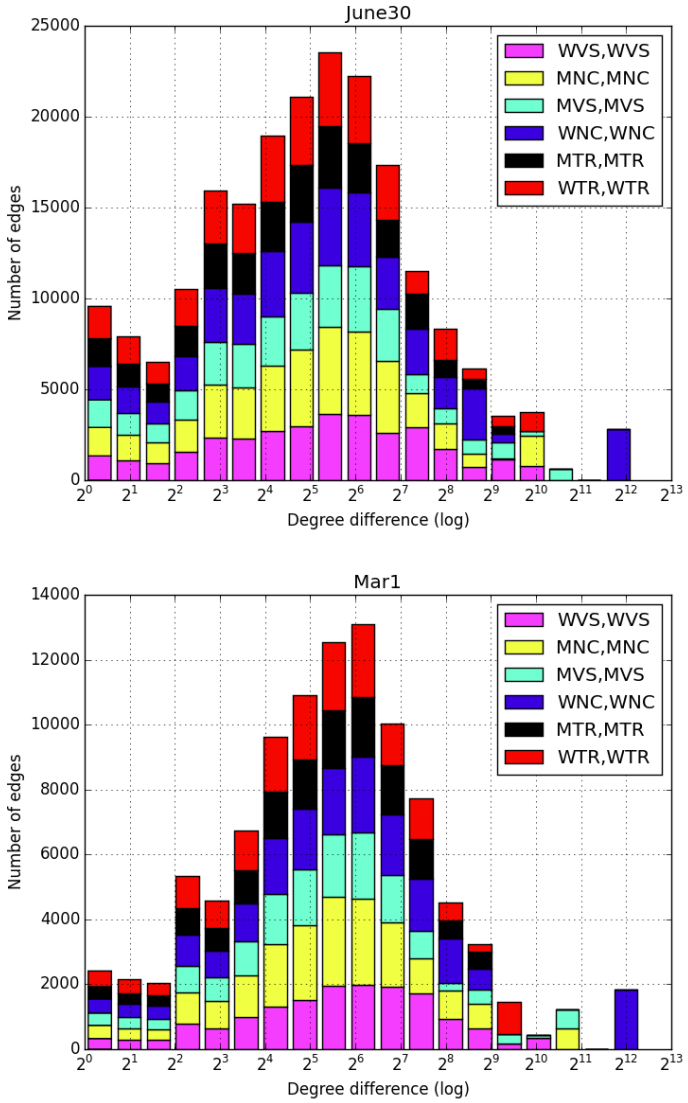


Figure 5.18: Degree differences of original avatars.

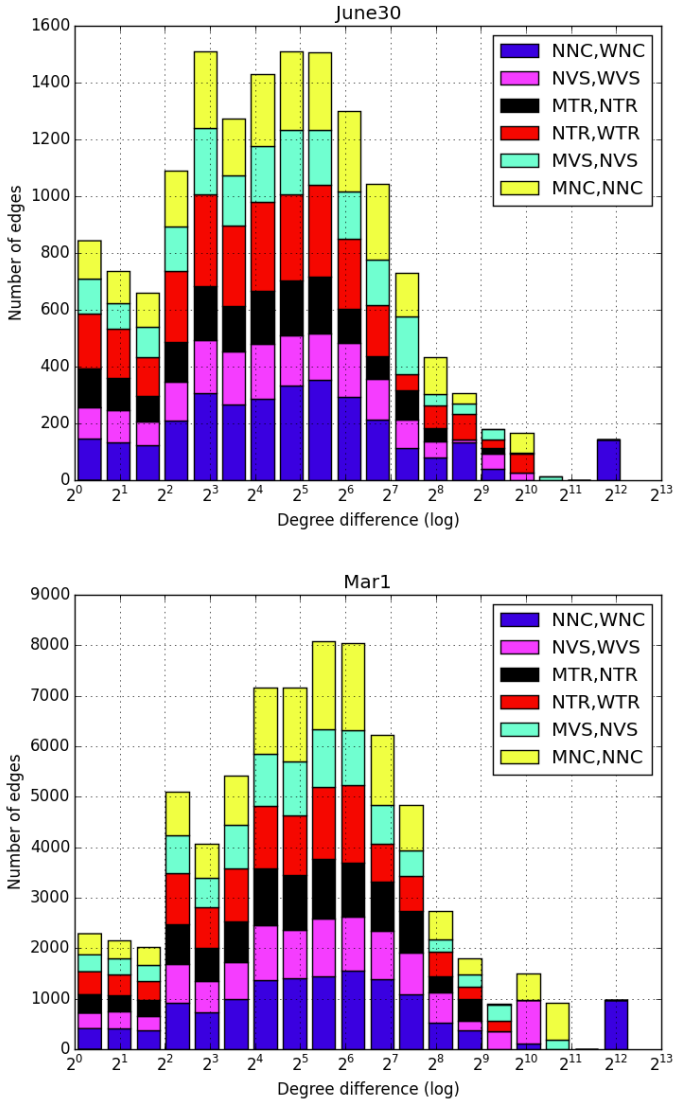


Figure 5.19: Degree difference from original to new avatars.

Discussion

It takes a surprising amount of time for the populations to actually start to mingle with each other. At least in terms of a community within a video game, while 7 months isn't a very long time in real life for our two example tribes to become indistinguishable. With a few notable exceptions, the lifespans of most games and their associated communities are not longer than 5 years.

This slow merger is especially interesting since these avatars have no way of knowing which server any other avatar is actually from. In conclusion the actual social cores of the original two networks have not so much merged as their core avatars have mutually bonded with newcomers, who replaced the original peripheral avatars.

Chapter 6

Graph motifs

This chapter covers our theoretical work on the local structure of a simple network model. The motivation being that a network model uses simple rules to generate a network and if we understand those rules we can identify what kind of local structures should appear in the final output. In this chapter we study the local structure of a graph created from the Barabási-Albert model through the lens of graph "motifs".

Definition 6.0.1. A *graph or network motif* are patterns that occur significantly more often than is expected in an ensemble of networks [21].

These graph motifs are a tool used to investigate the local structure of large complex networks. By comparing the frequency of small subgraphs in the real network to the frequency of the same subgraph in a ensemble of graphs they reveal local trends.

Motifs have been a popular area of study since they were introduced in [25]. The largest body of research has been focused on simply finding motifs efficiently, [17, 32] are two popular utilities that do just that. In order to determine if a particular subgraph is a motif of a network you need to find and count all unique induced subgraphs isomorphic to that subgraph. Then generate the ensemble of random graphs and do the same to each graph in the ensemble. This very quickly become a impossibly expensive problem for larger subgraphs and larger networks.

More recent research has shown bounds on the frequency of subgraphs in general graphs with a specific density [30]. This is what motivates this chapter.

Motifs background

Let n be the size of the motifs we are interested in. The *significance* of a motif is its z score. Let $|M|$ be the number of occurrences in our real network and $|\bar{M}_r|$ and σ_r be the mean and standard deviation of the number of occurrences in our random graphs. The significance is simply:

$$Z = \frac{|M_r| - |\bar{M}_r|}{\sigma_r}$$

.

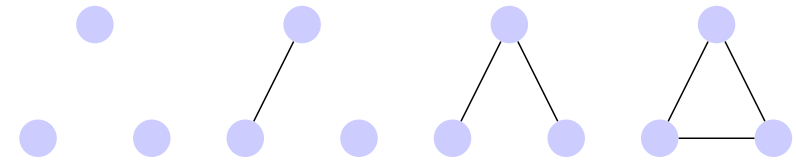
The collection of the z scores of all possible subgraphs of the same size n is the *significance profile*. Introduced in [21] the significance profile is the most common application of motifs we have seen in the literature, where they propose that the significance profile can be used as a sort of "graph fingerprint" typically used to group real networks together based on the similarities of their *normalized significance profile*

$$SP_i = \frac{Z_i}{(\sum_i \sqrt{Z_i^2})}$$

.

Motifs in online social networks

These graph fingerprints have been used in empirical case studies, including a study of the MMOG Pardus [28], the MMOG AION [27], and a study of Korean social network Cyworld [8].



(a) Empty $(1-p)^3$ (b) One edge $3p(1-p)^3$ (c) Open Triad $3p^2(1-p)$ (d) Triangle p^3

Figure 6.1: Examples of potential undirected subgraphs when $n = 3$.

When dealing with real world datasets most studies examined motifs of size 3 or 4 nodes like in Figure 6.1. In all three of these studies they found that the significance of sparse open triads and other subgraphs with few edges were low or negative while the significance of triangles cycles and cliques were positive. So triangles and the like are common motifs in other MMOG networks as well as online social networks generally.

See Figure 6.2 for a typical motif significance profile on 4 nodes from a snapshot. The data was collected using the Fanmod motif detection software. [32].

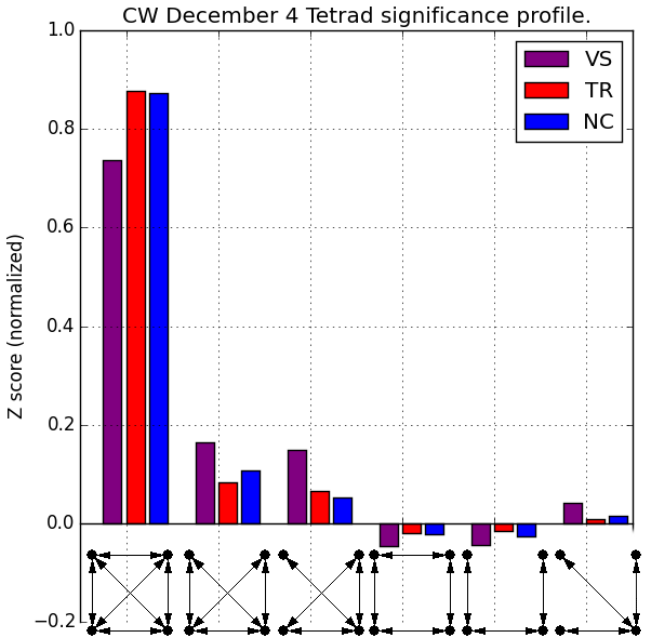


Figure 6.2: Motif significance profile from CW on Dec 4th.

Unfortunately the competition networks are too large for Fanmod to run

to completion. And unlike the planetside 2 data we cannot divide the network up into factions without potentially disrupting its local structure. Motif significance profiles are known to be highly dependent of the configuration of hubs included in the network [33].

Random graph ensemble

One of the most common ensemble used for motif detection is the random graphs that preserve the degree of the original network. The accepted method for creating such networks is the switching algorithm discussed in [20]. The algorithm is simple: choose a pair of edges from the graph uniformly at random and swap the endpoints unless it would create a multiedge or self loop. The algorithm preserves the degree of each of the four endpoints and after enough iterations it results in a random network. These ensembles are widely used in the literature and it is used in both Fanmod [32] and Kavosh [17] motif detection programs.

However, since the degree of each node is preserved the probability of a, edge between any given pair of nodes still depends on the degree of both endpoints. This introduced difficulty when comparing it to the probability of an edge in a given BA graph of finite size. So we will be working with an ensemble of $G_{n,p}$ random graphs to avoid degree dependence problems that occur when dealing with finite graphs with a unknown degree distribution.

In order to determine if a subgraph is a motif we need to know how many we should expect in the ensemble. Which is simply $\binom{N}{3}P$ where P is the probability of such a subgraph in a $G_{n,p}$ random graph $P = sp^{|E|}(p-1)^{|E^c|}$ where s is the symmetry factor for that graph, and E is the edge set. These are shown along with P in Figure 6.1.

The Barabási–Albert algorithm

The venerable Barabási-Albert created in 1991 was the first model to incorporate both the preferential attachment (rich get richer) mechanism and the network growth mechanisms observed in real world networks [3].

The algorithm takes two parameters, N the number of nodes in the final graph and m the number of new edges formed by each node added.

The probability that at time step t_v the new node v connects to existing node x is:

$$P(v) = \frac{k_x}{\sum\limits_{i \in V(G)} k_i} \tag{6.1}$$

The actual algorithm has two steps, first create a initial graph consisting of m nodes. Then while there are less than N nodes add a new node and m edges connecting it to m existing nodes chosen using the probability from

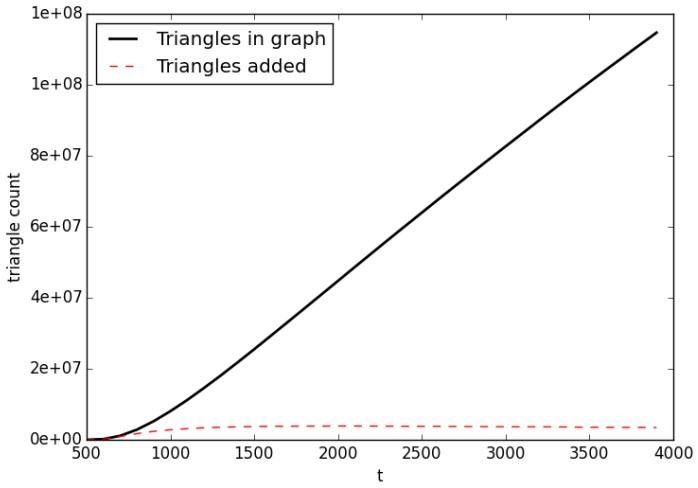
equation 6.1.

The configuration of the initial m nodes is intentionally left vague in [3], so we will start with an empty graph of m nodes with no edges.

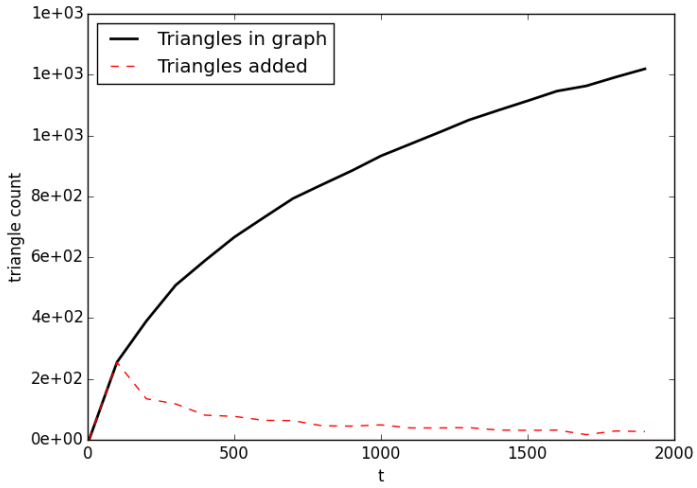
Decreasing triangle formation in empirical tests

The figures in this section are the result of empirical testing to see how many triangles are added as nodes are added. The test was done by running the BA algorithm counting the number of triangles added.

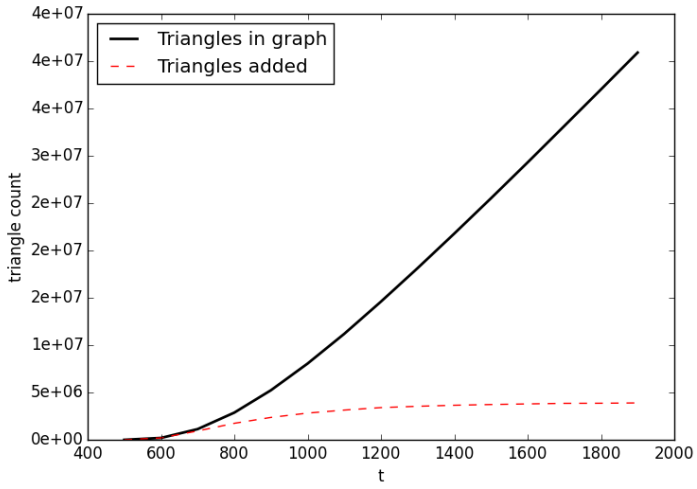
The first thing we notice is that if we continue to add nodes long enough the rate at which triangles are created eventually decreases. Since any give pair of nodes is progressively less likely to be connected, the only question is how large the difference between m and N will need to be before this happens.



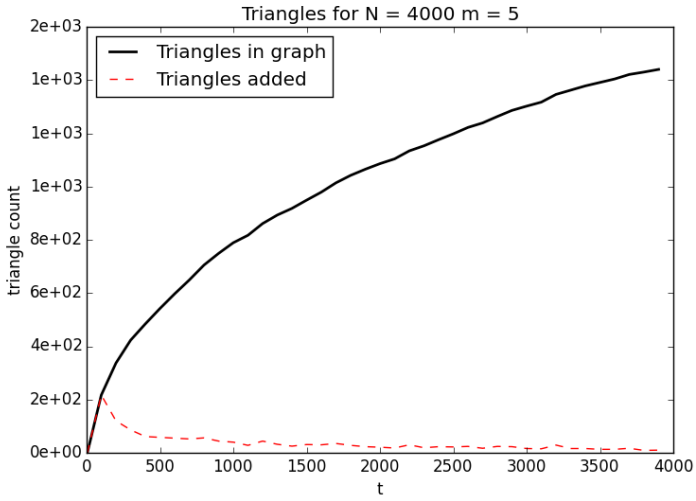
Triangles in BA: N = 4000 m = 500



Triangles in BA: N = 2000 m = 5



Triangles in BA: N = 2000 m = 500



Triangles in BA: N = 4000 m = 5

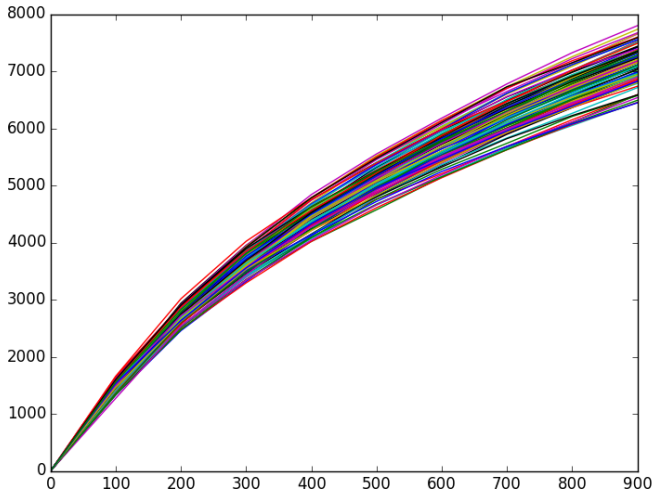


Figure 6.1: Triangles in BA: repeated with m = 5 N = 1000.

Figure 6.2: Triangles formed by nodes in BA algorithm

The plots in Figure 6.2 show the number of triangles created by 100 nodes

in as well as the net increase in triangles in red. For lower values of m it seems to follow a logarithmic trend, for higher values of m this does not hold for reasonably sized N .

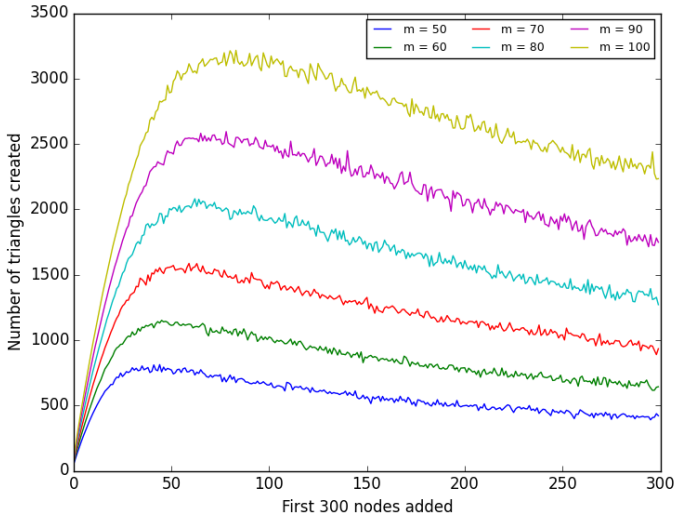


Figure 6.3: The triangles created by low values of m.

The Figure 6.3 shows the average number of triangles added by the first 300 nodes in 10 repetitions for a few low values of m. We conjecture that if we kept adding nodes to the graph eventually the number of triangles will eventually trend down.

Probabilistic bounds

This section covers some basic probabilistic properties we find a value for m such that the probability of a edge in a BA graph is greater than it is in the corresponding ensemble. Unless stated otherwise the formulas here can be found in [1].

The number of edges at time t depends entirely on the parameters N and m. Since we start from a empty graph and add m edges with every node the number of edges at any given time must be:

$$E(G_t) = m(t - m). \tag{6.2}$$

As a result many other graph parameters can be calculated at any given step such as density:

$$D = \frac{2m(t - m)}{t(t - 1)}. \tag{6.3}$$

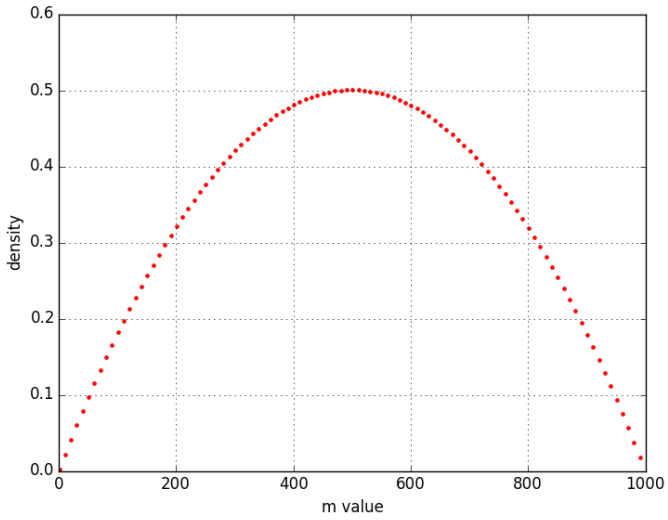


Figure 6.4: Density of a BA graph with 1000 nodes.

In order to determine whether or not a pair of neighbors of any given node are mutual (a triangle) or not (an open triad), we need a bound on probability of a edge between any two give nodes in our BA graph.

Without loss of generality suppose that node u is older than node v. So the probability that they are connected is determined by Equation 6.1 with the value of k_u at t_v at the time the node v was added. Which we cannot know since it dependent on when exactly t_u was as well as which nodes were chosen at every single time step between t_u and t_v.

Fortunately, the probability of attaching to a node v at any time t is bounded below by the probability of connecting to that node at any time t+1, this is a simple consequence of [1].

At each time step t $\sum_{i \in V(G)} k_i$ increases by exactly $2m$ while the degree of any given node increases by at most 1.

The probability that the node u will be chosen by the next node at time t is given by $P_t(u) = \frac{k_u}{2m(t-m)}$

Say u was chosen at time t the probability of u being chosen again at time t+1 is $\frac{k_u+1}{2m(t-m)+2m}$, if not it is $\frac{k_u}{2m(t-m)+2m}$. Thus

$$P_{t+1}(u) \leq \frac{k_u + 1}{2m(t-m) + 2m}$$

.

Trivially, $\frac{k_u+1}{2m(t+1-m)} \leq \frac{k_u}{2m(t-m)}$ with equality only when $k = N - m$.

Thus probability of an edge between any two in the BA graph is bounded below by $P_{ba} > \frac{m}{2m(N-m)}$.

While the probability of an edge in a $G_{n,p}$ random graph is equal to the density of the BA graph at time N, $D = \frac{2m(N-m)}{N(N-1)}$. This actually leads to $P_{ba} > \frac{m}{DN(N-1)}$ via substitution by the number of edges in the final graph. An example probabilities for BA graphs of 1000 nodes for every valid m value has been included in Figure 6.5.

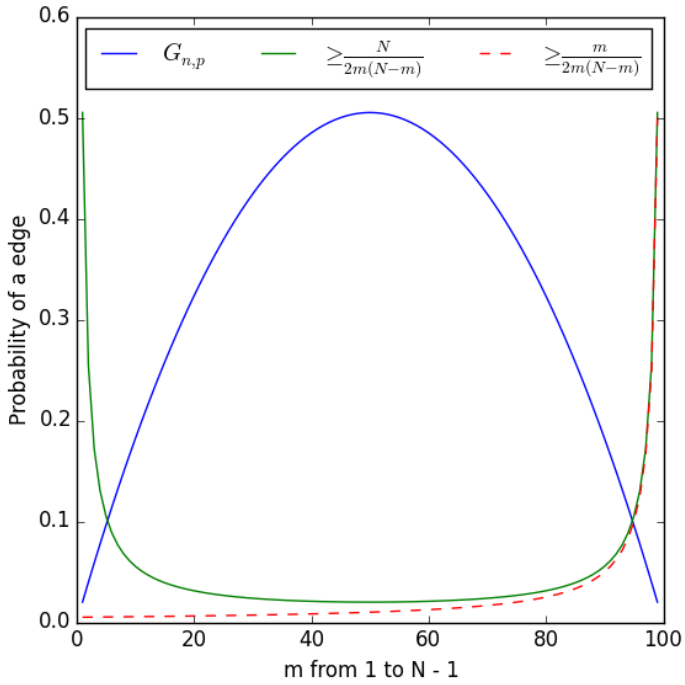


Figure 6.5: Examples of edge probabilities and bounds.

Figure 6.5 shows two minimal probabilities of a edge in the $G_{n,p}$ random graph. The absolute lower bound on the probability in the BA graph and the maximal lower bound on the probability of a edge connected to a degree N node.

This yields a probability we can substitute into the formula for the probability of a specific subgraph based on its edge configuration. The intercepts for $\frac{m}{2m(N-m)} = P_{G_{n,p}}$ are $N = \frac{8m^2 \pm \sqrt{16m^3 - 16m^2 + 1} - 1}{8m - 2}$.

This gives us the single valid intercept $\frac{8m^2 - \sqrt{16m^3 - 16m^2 + 1} - 1}{8m - 2}$ which is greater

than zero and less than N for all $m > 1$.

So when N is greater than $\frac{8m^2+\sqrt{16m^3-16m^2+1}-1}{8m-2}$ the probability of a edge is greater in the BA graph than in a $G_{n,p}$ graph.

Bounds from successive addition for $n = 3$

To get better results we have to look at a different perspective. If we were to simply count how many of each subgraphs are added with each new node to the graph we would know exactly how many we have in the end. This is impossible, but it is simple to find bounds on the number of subgraphs that can be added. It is also simple to find the expected value for the number in the ensemble. This gives us everything we need to produce bound on the motifs.

The expected number of any n node subgraphs in our ensemble which is simply $\binom{N}{3}P$, where P is the probability of such a subgraph in a $G_{n,p}$ random graph, as seen in eg. 6.1.

Additive bounds on cliques and empty graphs

For general bounds the maximum number of n cliques that can be added at any one step is at most $\binom{m}{n-1}$. In the final BA graph there will be exactly $\binom{N}{n}$ combinations of n nodes. Of those, $\binom{m}{n}$ will be be empty n node subgraphs

created by the initial m node subgraph. The number of combinations added with each node is.

$$\binom{a}{b} = \binom{a-1}{b} + \binom{a-1}{b-1}. \tag{6.4}$$

From Pascal's formula we know that at time t we add $\binom{t-1}{n-1}$ new combinations of n nodes to our graph. Thus

$$\binom{N}{n} = \binom{m}{n} + \sum_{t=m+1}^{N} \binom{t-1}{n-1}. \tag{6.5}$$

Of those exactly $\binom{t-1-m}{n-1}$ combinations do not include one of the m new edges that were added at time t. This gives an upper bound on the number of empty ($|E_0|$) n node subgraphs added at each t:

$$|E_0| < \binom{m}{n} + \sum_{t=m+1}^{N} \binom{t-1-m}{n-1}$$

So the number of n subgraphs that at least one edge ($|E_1|$) must be:

$$|E_1| > \binom{N}{n} - \left(\binom{m}{n} + \sum_{t=m+1}^{N} \binom{t-1-m}{n-1}\right) \tag{6.6}$$

$$|E_1| > \sum_{t=m+1}^{N} \binom{t-1}{n-1} - \binom{t-1-m}{n-1} \tag{6.7}$$

Finding bounds 3 node directed subgraphs

In this section we use the directed BA model with $n = 3$.

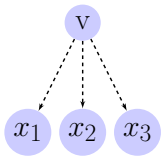
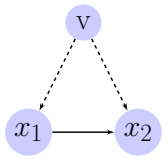


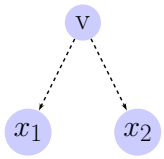
Figure 6.6: Time step.

At each step we add a new node with edges to m existing nodes, we will call the new node v and the existing neighbors $x_1, x_2, ..., x_m$ as in eg. 6.6.

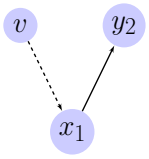
The directed BA algorithm can only create 4 connected directed graphs on three nodes. These are one of the directed triangles and three potential open triads. These valid configurations are shown in Figure 6.7 the dashed lines indicate the edges added by the new node v.



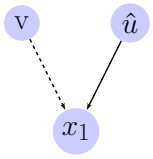
Triangle ($\triangle$).



Type 1 path (P1).



Type 2 path (P2).



Type 3 path (P3).

Figure 6.7: Possible sub graphs.

First at every time step t_v the new node v adds a combined total of exactly $\binom{m}{2}$ new triangles and type one paths. So $|\triangle| + |P1| = \binom{m}{2}$ and since this is the only to add these subgraphs $|P1| \leq \binom{m}{2}, |\triangle| \leq \binom{m}{2}$.

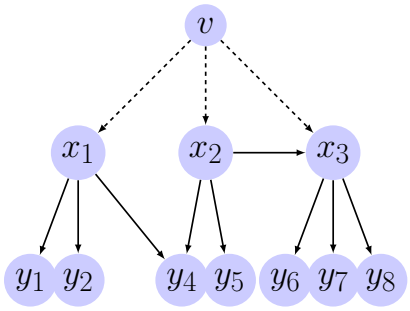


Figure 6.8: Depth 2 tree example.

As illustrated by Figure 6.8, there are m^2 potential type 2 paths added with each new node, but every triangle created by v reduces that count by one, so $|P2| = m^2 - |\triangle|$. If x_i was one of the m nodes in the initial empty

graph, it would have out degree zero [1] , so it is possible to create anywhere from 0 to m^2 type two paths.

That leaves us with the type 3 path, the number of which depends on the in degree [2] of the new neighbors $x_1, x_2, ..., x_m$, eg. 6.7. Since the BA algorithm will be likely to choose high in degree nodes the number of P3 can be quite high.

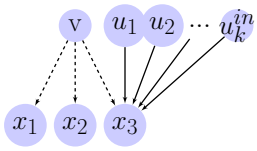


Figure 6.9: Type 3 path example

So $|P3| = \sum_{i \in neigh(v)} k_{in}(i) - |\triangle_i|$ which is bounded above by $m(t_v - 1)$ the case where all nodes choose the inital m nodes every time. The lower bound is of course 0 since the existence of a single triangles implies thet one neighbour must have a incomeing edge existence of an incoming edge eg. 6.10.

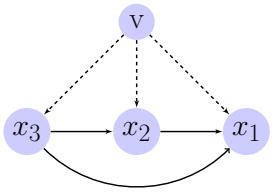


Figure 6.10: Least possible type 3 path configuration.

So the number of undirected paths ($|\bigwedge|$) is the combined number of

[1] Number of outgoing edges.
[2] Number of incoming edges.

directed paths:

$$|\bigwedge| = \sum_{t=m}^{N} \binom{m}{2} + m^2 + \sum_{i \in neigh(v)} k_{in}(i) - 3|\triangle_t|$$

$$|\bigwedge| > (\binom{m}{2} + m^2 + -2|\triangle_t|)(N - m)$$

With the bound on triangles this gives:

$$|\bigwedge| > (m^2 - \binom{m}{2})(N - m)$$

$$|\bigwedge| > \frac{1}{2} m(m + 1)(N - m)$$

It is a lower bound on the number of paths or open triads.

Connected Triads

Figure 6.11 shows the expected number of open triad and triangle sub graphs in a $G_{n,p}$ vs the bounds on the same in a BA graph for $N = 1000$ and all valid values of m.

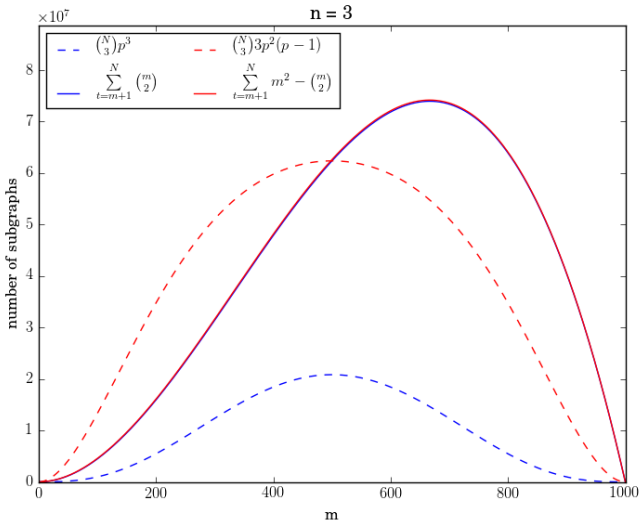


Figure 6.11: Open triad and triangle subgraph examples.

We now determine results for motifs of size three using our additive bounds.

Triangles

As we know at each timestep $|\triangle| \leq \binom{m}{2}$. So we can get a bound on triangles in the final BA graph easily. So

$$\sum_{t=m+1}^{N} \binom{m}{2} = \frac{1}{2}m(m-1)(N-m) \tag{6.8}$$

Is the upper bound on the number of triangles in any BA graph.

The only place the triangle can be motif is where the expected number of triangles in the ensemble of random graphs is less than this upper bound. Though in reality our empirical tests lead us to expect that number of triangles created at each step to decrease as t increases. So as $m \to N$ the number of triangles should be higher.

The expected number of triangle subgraphs in a $G_{n,p}$ random graph is:

$$\binom{N}{3} p^3 = \frac{4}{3} \frac{(N-2)(N-m)^3 m^3}{(N-1)^2 N^2}. \tag{6.9}$$

Trivially:

$$\frac{1}{2} m(m-1)(N-m) > \frac{4}{3} \frac{(N-2)(N-m)^3 m^3}{(N-1)^2 N^2}$$

For all $0 < m < N$. So our upper bound on the number of $\triangle$ in the BA graph is always less than the number of triangles in expected the random graph ensemble.

Open Triad

Now we apply the bound on open triad subgraphs. The expected number of open triads in a $G_{n,p}$ random graph is given by.

$$\binom{N}{3} 3p^2(p-1) = \frac{2(N-2)(N-m)^2m^2(N^2-2Nm+2m^2-N)}{(N-1)^2N^2}. \quad (6.10)$$

Now we need the intercept:

$$\frac{1}{2}m(m+1)(N-m) \geq \frac{2(N-2)(N-m)^2m^2(N^2-2Nm+2m^2-N)}{(N-1)^2N^2}. \quad (6.11)$$

This becomes an equality when $m \approx N/2 - 1$. Which can be shown via substituting $m = \frac{N}{2} - 1, \frac{N}{2} - 2$ into the inequality 6.11. When $m = \frac{N}{2} - 1$ the inequality holds $\forall N \geq 2$. If $m = N/2 - 2$ the inequality only holds $\forall N < 2$. Therefore for any valid m and N where $N > 2$ the open triad will always be a motif when $m \geq \frac{N}{2} - 1$.

Disconnected Triads

One Edge

Earlier we found the general minimum number of subgraphs containing a single edge. In equation 6.6 the case of a three node subgraphs gives:

$$\sum_{t=m+1}^{N} \binom{t-1}{2} - \binom{t-1-m}{2} = \frac{1}{2}m(N-2)(N-m). \tag{6.12}$$

The expected number of 3 node subgraphs containing exactly one edge in a $G_{n,p}$ random graph is:

$$\binom{N}{3} 3p(p-1)^2 = \frac{(N-2)(N-m)m(N^2-2Nm+2m^2-N)^2}{(N-1)^2N^2}. \tag{6.13}$$

All we need to do is find the intersection between Equation 6.12 and Equation 6.13 which simplifies to:

$$\frac{1}{2} = \frac{(N^2-2Nm+2m^2-N)^2}{N^2(N-1)^2}$$

.

This has two intercepts under the conditions of $1 < m < N$:

$$m = \frac{1}{2}N + \frac{1}{2}\sqrt{(\sqrt{2}-1)N^2 + (2-\sqrt{2})N} \tag{6.14}$$

$$m = \frac{1}{2}N - \frac{1}{2}\sqrt{(\sqrt{2}-1)N^2 + (2-\sqrt{2})N} \tag{6.15}$$

So the maximum number of one edge subgraphs in the BA graph is greater than the expected number in the $G_{n,p}$ only when $m >$ Equation 6.14 or $m <$ Euation 6.15 for any $N > m > 0$.

Empty

And finally for the empty subgraph for the upper bound on the number of empty nodes in a BA graph when $n = 3$ the bound is:

$$\binom{m}{3} + \sum_{t=m+1}^{N} \binom{t-1-m}{2} = \frac{1}{6}(N-2)(N^2 - 3Nm + 3m^2 - N) \tag{6.16}$$

The expected number of empty graphs in a $G_{n,p}$ graph is:

$$\binom{N}{3}(1-p)^3 = \frac{1}{6}\frac{(N-2)(N^2 - 2Nm + 2m^2 - N)^3}{(N-1)^2N^2} \tag{6.17}$$

This, for all $N > 5$, becomes an equality only when $m = 0$ and $m = N$, neither of which are in our domain. See Figure 6.12 for another example showing the expected number of single edge and empty sub graphs in a $G_{n,p}$ vs the bounds on the same in a BA graph for $N = 1000$ and all valid values of m.

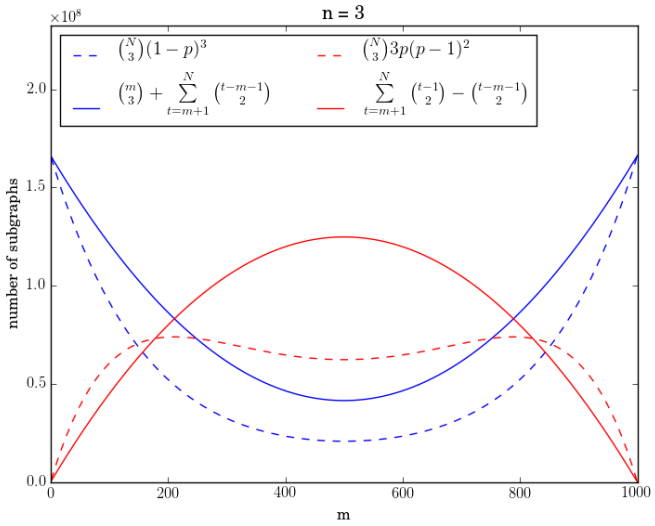


Figure 6.12: One edge and empty subgraph example.

Final bounds

So in conclusion the final bounds for the simple case of a 3 node motif in a BA graph are as follows. Whenever

$$N \geq \frac{8m^2 - \sqrt{16m^3 - 16m^2 + 1} - 1}{8m - 2}$$

holds then for a sufficiently large ensemble of random graphs, the triangle will be a motif and an empty subgraph will never be a motif.

The open triad will always be a motif whenever:

$$m > \frac{N}{2} - 1$$

If $N > 2$. And finally the one edge subgraph can only be a motif when:

$$m > \frac{1}{2}N + \frac{1}{2}\sqrt{(\sqrt{2}-1)N^2 + (2-\sqrt{2})N}$$

or

$$m < \frac{1}{2}N - \frac{1}{2}\sqrt{(\sqrt{2}-1)N^2 + (2-\sqrt{2})N}$$

for any valid N.

Chapter 7

Conclusions

In the end the actors that were removed or left in each network are in a way similar, mainly being minor players in unimportant positions. Despite the purpose of each network being different (competition vs. cooperation), the removed nodes were similar. They were those actors that were least engaged in the task at hand, if they had been more interested they would have had advertisements appearing under more than a handful of search terms, had they been having a bit more fun they would probably have had more friends.

This is why it is so difficult to say with certainty that the removal process is anything but random. Since random node removal on power law networks is more likely to remove these types of nodes because of there abundance while hubs remain simply by virtue of rarity. In general if we had a network where removal process was favoring the opposite and removing the key actors

how could the network be survive and grow to become a large network?

Conclusions

In our PlanetSide 2 data we find that peripheral nodes are most likely to leave, and that in turn central nodes are most likely to stay around. However we cannot say which way this relation runs, are hubs more likely to carry on because of social links or do they have social links because they are unlikely to leave. We should not presume that the network drives the likelihood of death as we cannot tell from the data we have which way this runs and likely its a bit of both depending on the individual in question.

While peripheral nodes are most likely to be removed are they on the periphery because they are less interested in general or are they in the periphery because nobody brought them in?

Likewise in our advertisement networks we see that closed business are isolated from one another and not central, but this can come from the other way.

Future Work: Additional database analysis.

By its very nature large datasets will always have more unanswered questions. There are a huge number of potential relationships between the networks, the actors and their removal that we did not have time to test, for example how many of the removed avatars had a typo in their name. In this suggests some future work that seems interesting but is either outside the scope of large network analysis or simply something we did not have time to do.

The formation of a social network

Earlier this summer PlanetSide 2 was relaunched on the playstation 4 home console to a new market and player base. We collected a large number snapshots of all 5 of the launch servers for the first few weeks.

The launch was not particularly successful and as of the time of writing of the original 15 or so servers only 1 remains. Regardless this presents a opportunity to watch the early development for this sort of network. And as we have seen the first few weeks is when the majority of hubs are added.

Identifying Alternate avatars

In MMOGs it is common for players to have a pool of alternative avatars (alts) in addition to there primary or main avatar. That they play less frequently for various reasons such as experiencing other factions or socialize with players on the other servers.

Identifying these avatars may be possible and it would allow us to map the avatar friendships back into a better approximation of the network of actual player relationships. It is entirely possible that the three lobed faction structure might collapse into a single mass under such a mapping. It also gives a natural definition for strong links as people who play linked together across multiple avatars, which would open up new analysis methods.

The common pool class items that when purchased with would be are unlocked across all avatars on the account. Some players opt to have a naming scheme across their all of there alternates. So alts could potentially be found by analyzing names so matching algorithm could potentially find them. For example TheDestroyerOfHats,TheHarbingerOfHats and TheApotheosisOfHats are my main and two alts respectively. Some outfits also maintain a altfit these are simply a formal outfit for their alts on a different faction or server.

Chapter 8

Appendix 1: Code

Api crawler

Snapshot algorithm

To run the provided code simply create a main_data_crawler object it takes the initial of the server you want as a its only parameter, then simply call run() and it will attempt to gatherer and record a snapshot for you. The crawler has two sections that work as follows.8.

Part one: Crawling the friend lists

1. The first step requires a list of active avatar Ids from the server we want to crawl. 8 pulls some avatars from the leader boards [1]. But in the original we just used a few random avatars Id's found manually. Add these seeds to the queue of Ids to check.

2. Get the friendlists of all avatar Ids in the queue from the API. If the friendlist of a Id is successfully found remove it from the queue and add it to the done list of visited avatar Ids.

3. Then go through each friend in the friend list. Each friend entry also includes the both the last login date and server Id, this makes it easy to sort the valid Ids that are active in the last 44 days and the correct server. Save each of these valid friend relationships to the edge set. If they are not in the list of Ids that have been checked already add each of these valid Ids to the list of Ids to check.

4. If there are still Ids in the queue of Ids to check return to step two otherwise continue.

5. Terminate and the API connection and record the edge list as a sql table.

[1]The top 100 avatars for kill count and death count that week

Part two: Get avatar attributes

The second phase gets the node attributes, this is done separately from the crawler for the sake of simplicity. Because the information comes from four separate collections that must be joined. [2]

1. Get every avatar Id found by our crawl in part one break it up into small chunks in order to comply with API restrictions.
2. Then query the API about each batch of avatar Ids and get the attributes of those avatars.
3. Unpack the avatar attributes from the response into a new sqlite table.

The actual code is a more complex since we need to handle failed API calls, check data integrity and comply with the API call as well as record it into our sqlite database. For more detail see the comments in 8. Note that this code was revised to store information directly into the sqlite database, instead of text files.

It also archives the results of every successful response from the API this means that the crawl can be canceled and resumed later without starting from scratch and hammering the API with repeated calls. All of the data used in this paper was gathered with the older functionally identical but clunky version.

[2]it takes significantly longer than the first due to restrictions on API calls.

Sql database structure

We used the sqlite3 package with python 3.4.3 to construct and manipulate all of the sqlite databases copies of each database are available on request. The three original databases are Connery.db, Emerald.db and Miller.db. The data for a snapshot is recorded as a table of edges and edge attributes and a second table of node attributes. The table names are simply the month and day the crawl was run. So for example the snapshot recorded on the fourth of August 2014 would be saved in two tables 'Aug4e' and 'Aug4' containing the edge set and node attributes respectively.

Source code

PS2 API Crawler

9pt

```
'''
Rewritten for readability but it keeps it original loop structure, it
would be dishonest to change how I did it for the original data.

Read through everything carefully the structure is very odd.

This code builds a graph of the frendslist of active characters in the
    MMO planetside 2 by using the census API.

Information on the API is found here at http://census.soe.com/, you
   may request a Service ID of your own on the page they seem very

The official limit on query's is no more than 100 in 1 minute, or you
   risk having your connection terminated or being banned outright from
    the API.
'''
import sqlite3
import json
import os
import urllib.request
import time
import datetime
import networkx as nx

def singleCol(conn,query):
    return [i[0] for i in conn.execute(query).fetchall()]

def multiCol(conn,query):
    return [list(i) for i in conn.execute(query).fetchall()]

'''
The Playstation 4 has 2 separate name spaces one for Europe and one
   for north America the correct name space must be used for each
   servers,
those name spaces as well as the world Id numbers are contained in the
```

```
    following dictionaries keyed by the initials of the server you want
    to investigate.
US: ps2ps4us:v2
    Servers
    Palos world_id = 1001 Initial: P
    Genudine world_id = 1000 Initial: G

EU: ps2ps4eu:v2
    Servers
    Ceres world_id = 2000 Initial: Ce
    Lithcorp world_id = 2001 Initial: L

The PC name space is ps2:v2
    Servers
    Connery world_id = 1 Initial: C
    Emerald world_id = 17 Initial: E
    Miller world_id = 10 Initial: M
'''
namespace_dict = {'P':'ps2ps4us:v2','G':'ps2ps4us:v2','S':'ps2ps4us:v2
    ','Cr':'ps2ps4us:v2','Ce':'ps2ps4eu:v2','L':'ps2ps4eu:v2','C':'ps2:
    v2','E':'ps2:v2','M':'ps2:v2'}
server_id_dict = {'G':'1000','P':'1001','Cr':'1002','S':'1003','Ce':'
    2000','L':'2001','C':'1','E':'17','M':'10'}
server_name_dict = {'P':'Palos','G':'Genudine','Ce':'Ceres','S':'
    Searhus','L':'Lithcorp','C':'Connery','E':'Emerald','M':'Miller','Cr
    ':'Crux'}
# Contains all the methods settings etc needed to build the friend-
    list graph and fetch node attributes of a server in planetside2. The
    class takes the initial of the server to make a graph of when it is
    first called.
class main_data_crawler():
    def __init__(self,server_inital):

        # This limits strain on the database by restricting our
    attention to only those nodes active in last 44.25 days.
        self.curTime = time.mktime(time.localtime())
        self.DT = datetime.datetime.now()
        self.tSinceLogin = self.curTime - 3824794
        # Using the dictionaries above get the name space server Id
```

```
    and server name of the server we wish to crawl. Remember you can
    change these class variables if needed.
        self.namespace, self.server_id, self.server_name =
    namespace_dict[server_inital], server_id_dict[server_inital],
    server_name_dict[server_inital]
59      # The table name is a combination of the server name plus the
    current month and day
        self.table_name = self.server_name+self.DT.strftime("%B")+str(
    self.DT.day)
61
        # The set done contains every node we have already examined
    and either added to the data or rejected as too old or otherwise
    invalid.
63      self.done = set()

65      # idDict stores the data we have collected the key is the Id
    of the node while the values are the values of its friends.
        self.idDict = {}
67
        # I have two computers and thus two paths to the Dropbox. You
    will want to replace self.mypath with whatever path you want to use
    for storing your data.
69      if os.path.exists('D:\\Dropbox\\PS2Research\\'):
            self.mypath = 'D:\\Dropbox\\PS2Research\\NewData\\'
71      else:
            self.mypath = 'C:\\Users\\John\\Dropbox\\PS2Research\\
    NewData\\'
73
        # The archive database saves the responses from the API so no
    API call is ever done twice, this is new this version.
75      self.archive = sqlite3.connect(self.mypath+'\\archive.db')
        # We have two types of data we gather type one is a friends
    list and the node attributes
77      self.archive.execute('Create table if not exists '+self.
    table_name+'Edge (Id Primary key,raw TEXT)')
        self.archive.execute('Create table if not exists '+self.
    table_name+'Node (Id Primary key, raw TEXT)')
79
        # The seed_node table records the set of seed nodes just in
```

```
    case it is needed for debugging or some unforeseen purpose.
81          self.archive.execute('Create table if not exists seed_nodes (
    name TEXT, seed_nodes TEXT)')

83          # This database stores the unpacked data in the format used
    through my analysis code.
            self.database = sqlite3.connect(self.mypath+self.server_name+'
    .db')
85          # the data in two tables Eset and Node which have the format
    my code actually uses and a third table history stores stat history
    data in case I want to do something with that later.
            self.database.execute('Create table if not exists '+self.
    table_name+'Eset (Source TEXT, Target TEXT, Status TEXT)')
87          self.database.execute('Create table if not exists '+self.
    table_name+'Node (Id PRIMARY KEY, name TEXT, faction TEXT, br INTEGER,
    outfitTag TEXT, outfitId INTEGER, outfitSize INTEGER, creation_date
    INTEGER, login_count INTEGER, minutes_played INTEGER ,last_login_date
    INTEGER, kills INTEGER, deaths INTEGER)')
            self.database.execute('Create table if not exists '+self.
    table_name+'History (Id PRIMARY KEY, history TEXT)')
89
            # Get the starting nodes from the leader-boards.
91          # If we already have seed nodes for the day simply retrieve
    them, otherwise gather some.
            if self.table_name in singleCol(self.archive,'SELECT name from
     seed_nodes'):
93              seed = singleCol(self.archive,'SELECT seed_nodes from
    seed_nodes where name = "'+self.table_name+'"')[0]
                self.listToCheck = seed.split(',')
95          else:
                self.listToCheck = self.leader_board_sample(75)
97
     # Gathers edges then gathers information on the nodes.
99      def run(self):
            print('Gathering edges')
101         self.get_friendlist_network()
            print('Gathering node attributes')
103         self.get_node_attributes()
```

```
105     # Crawls the friend list network.
        def get_friendlist_network(self):
107         # Get raw responses from the server for the nodes we got from
    the leader board earlier.
            self.get_friends(self.listToCheck)
109         while len(self.listToCheck)>0:
                self.expand_graph()
111             print('Nodes left to check '+str(len(self.listToCheck)))

113     # Updates the list of nodes to Check and manages what nodes to ask
        the API about, also saves data to the SQL database.
        def expand_graph(self):
115         Queue,Check,add_to_queue = [],[],[]
            # Get the list of nodes we have server responses for.
117         valid = list(self.idDict.keys())
            for i in self.listToCheck:
119             # If i is in the id_dict we unpack its friends-list and go
     through each of its friends to determine if we traverse them as
    well. Other wise we add it to the Queue.
                if i in valid:
121                 friend_list = self.idDict[i]
                    self.done.add(i)
123                 total_friends = len(friend_list)
                    for friend in friend_list:
125                     Id = friend.get('character_id',-1)
                        world_id = friend.get('world_id',-1)
127                     last_online = int(friend.get('last_login_time',-1)
    )

129                 # now we write all edges to our database with the
    following rules.
                    # Friends that are in self.done are ignored since
    those edges have already been added.
131                 # Friends who have not logged on since self.
    tSinceLogin are given old status
                    # Friends who are on different servers are given
    cross server status
133                 # Friends who are both on different servers and
    have not logged in since are called both
```

```
135                     # Only valid nodes that are not in done are added to the Queue.
                        if Id not in self.done and last_online > self.tSinceLogin and int(world_id) == int(self.server_id):
137                         status = 'normal'
                            Queue.append(Id)
139                     elif last_online < self.tSinceLogin and world_id != self.server_id:
                            status = 'both'
141                         self.done.add(Id)
                        elif last_online < self.tSinceLogin:
143                         status = 'old'
                            self.done.add(Id)
145                     elif world_id != self.server_id:
                            status = 'cross server'
147                         self.done.add(Id)
                        else:
149                         status = 'error'
                            self.done.add(Id)
151                     # Inserts the information into the database in the expected format.
                        self.database.execute('INSERT INTO '+self.table_name+'Eset (Source,Target,Status) Values(?,?,?)',(i,Id,status))
153             else:
                    Check.append(i)
155                 Queue.append(i)
            Check,Queue = list(set((Check))),list(set((Queue)))
157         print('Queue len '+str(len(Queue)))
            print('Query len '+str(len(Check)))
159         # If there are nodes in check, then get their friends list from the server.
            if len(Check) > 0:
161             self.idDict = self.get_friends(Check)
            # Then add those nodes to the list of nodes to check.
163         self.listToCheck = Queue
            self.database.commit()
165
```

```
    # Returns a dictionary where the keys are Ids and values are their
    friends lists
    def get_friends(self,to_check):
        print('Gathering friendlists')
        start_time = time.mktime(time.localtime())
        # Load existing values
        idDict = {}
        # All nodes we have a archived friends-list for already.
        archive_id = singleCol(self.archive,'Select Id from '+self.
    table_name+'Edge')
        # List of all nodes for which no archived value exists.
        remaining_nodes = [n for n in to_check if n not in archive_id]
        for l in self.chunks(remaining_nodes,40):
            url = 'http://census.soe.com/s:GraphSearch/get/'+self.
    namespace+'/characters_friend/?character_id='+','.join(l)+'&c:
    resolve=world&c:show=character_id,world_id'
            time.sleep(2.0)
            jsonObj =urllib.request.urlopen(url)
            decoded = json.loads(jsonObj.read().decode('utf8'))
            results = decoded['characters_friend_list']
            for x in results:
                # First dump the raw results of the call into a
    archive, the first column.
                try:
                    self.archive.execute('Insert into '+self.
    table_name+'Edge (Id,raw) VALUES(?,?)',(x['character_id'],json.dumps
    (x)))
                except:
                    # Usually when/if this fails its because the
    server is down.
                    print('archive failure')
                    if 'error' in str(decoded):
                        print('Server down')
                        exit
                    else:
                        raise
            for f in results:
                idDict[f['character_id']]=f['friend_list']
            self.archive.commit()
```

```
197         # Load in the friends-list from any archived results we may
    already have.
            archived_friends_lists = self.sql_columns_to_dicts('Edge','raw
    ',self.archive)
199         for l in [i for i in to_check if i not in remaining_nodes]:
                f = json.loads(archived_friends_lists[l])
201             try:
                    idDict[f['character_id']]=f['friend_list']
203             except:
                    print('get friends error')
205                 print(l)
                    print(f)
207                 raise
            print('Elapsed time: '+str((time.mktime(time.localtime())-
    start_time)))
209         return idDict

211     # Gathers all node attributes.
        def get_node_attributes(self):
213         edges = multiCol(self.database,'SELECT Source,Target from '+
    self.table_name+'Eset where Status="normal"')
            G = nx.Graph()
215         G.add_edges_from(edges)
            self.getCharData(G.nodes())
217         self.interp_character_data()

219     # Gets character attributes for each node in the graph formed from
        the edges.
        def getCharData(self,nodes):
221         # Check for any nodes already added to the archive, does not
    query those nodes again saving time and bandwidth.
            archive_id = singleCol(self.archive,'Select Id from '+self.
    table_name+'Node')
223         remaining_nodes = [n for n in nodes if n not in archive_id]
            re_count = len(remaining_nodes)
225         print('Number of nodes in graph is: '+str(len(nodes))+' Number
     of unarchived nodes is: '+str(re_count))
            # Break the list up into chunks of 40
227         smallLists = self.chunks(remaining_nodes,40)
```

```
        i = 0
229     for l in smallLists:
            search = ','.join(l)
231         # After 5000 iterations print the progress % just so we
    know it isn't frozen.
            if i%5000 == 0:
233             print('looking up data completion is at '+str(100*(i/
    re_count))+'%')
            url = 'http://census.soe.com/s:GraphSearch/get/'+self.
    namespace+'/character/?character_id='+','.join(l)+'&c:resolve=outfit
    ,name,stats,times,stat_history'
235         jsonObj =urllib.request.urlopen(url)
            decoded = json.loads(jsonObj.read().decode('utf8'))
237         results = decoded['character_list']
            for x in results:
239             # Unpack the server response and add each to the
    archive.
                try:
241                 self.archive.execute('Insert into '+self.
    table_name+'Node (Id,raw) VALUES(?,?)',(x['character_id'],json.dumps
    (x)))
                except:
243                 print('archive failure')
                    if 'error' in str(decoded):
245                     print('Server down')
                        exit
247                 else:
                        raise
249         self.archive.commit()
            i = i +40
251         # The 2 second wait seems to be enough to avoid hitting
    the soft limit of API calls.
            time.sleep(2.0)
253

     # Simply unpacks the character data gathered previously, reading
    the raw data from the archive and writing it into the database.
255  def interp_character_data(self):
        completed_id = singleCol(self.database,'SELECT Id from '+self.
    table_name+'Node')
```

```
257         results = []
            for raw in multiCol(self.archive,'SELECT Id,raw from '+self.
    table_name+'Node'):
259             if raw[0] not in completed_id:
                    results.append(json.loads(raw[1]))
261         # Unpack and add it to the snapshots.
            for x in results:
263             try:
                    # Basic avatar information.
265                 Id = int(x.get('character_id','0000000000000000000'))
                    name = x['name'].get('first','not available')
267                 faction_id = x.get('faction_id',-1)
                    faction = {'1':'VS','2':'NC','3':'TR'}.get(faction_id,
    'has no faction')
269                 br = x.get('battle_rank',{'value':'-1'})['value']
                    # Time data:
271                 t = x.get('times')
                    creation_date = t.get('creation','0')
273                 login_count = t.get('login_count','0')
                    minutes_played = t.get('minutes_played','0')
275                 last_login_date = t.get('last_login','0')
                    # Outfit data:
277                 o = x.get('outfit',{'placeholder':'error2'})
                    outfitTag = o.get('alias',-1)
279                 outfitName = o.get('name','not available')
                    outfitId = o.get('outfit_id',-1)
281                 outfitSize = o.get('member_count',-1)
                    # Stat history is formatted differently, it returns a
    list of stats of dictionaries containing the stat history:
283                 stats = x.get('stats',{'placeholder':'error2'}).get('
    stat_history')
                    if type(stats) == list:
285                     # If they add more stats the order of the deaths
    and kills will likely change so these indices's would need to be
    changed.
                        D,K = stats[2],stats[5]
287                     if D.get('stat_name') != 'deaths' or K.get('
    stat_name') != 'kills':
                            print(Id)
```

```
289                         # print(stats)
                            # break
291                     kills = K.get('all_time',-1)
                        deaths = D.get('all_time',-1)
293                 else:
                        kills,deaths = -1,-1
295                 self.database.execute('INSERT or replace INTO '+self.
table_name+'History (Id,history) VALUES(?,?)',(Id,json.dumps(stats))
)
                    long_sql = '(Id,name,faction,br,outfitTag,outfitId,
outfitSize,creation_date,login_count,minutes_played,last_login_date
, kills, deaths) Values(?,?,?,?,?,?,?,?,?,?,?,?,?)'
297                 self.database.execute('INSERT or replace INTO '+self.
table_name+'Node '+long_sql,(Id,name,faction,br,outfitTag,outfitId,
outfitSize,creation_date,login_count,minutes_played,last_login_date
,kills,deaths))
            except:
299                 raise
        self.database.commit()
301

    # I have used more than one method to get the initial list of
character Ids. The first version simply used the id's of characters
I was knew of.
303     # This new version is a bit less biased It gathers the players who
     were in the top limit places on the current leader-board for all
areas of the leader-board available.
    # Note that all leader-board stats are strongly correlated.
305     def leader_board_sample(self,limit = 50):
        url_start = 'http://census.soe.com/s:GraphSearch/get/'+self.
namespace
307         seed_ids = []
        for leaderboard_type in ['Kills','Time','Deaths','Score']:
309             url = url_start+'/leader-board/?name='+leaderboard_type+'&
period=Forever&world='+self.server_name+'&c:limit='+str(limit)
            jsonObj =urllib.request.urlopen(url)
311             decoded = json.loads(jsonObj.read().decode('utf8'))
            try:
313                 H = decoded['leaderboard_list']
            except:
```

```
315                 print(decoded)
                    print(url)
317             for characters in H:
                    Id = characters.get('character_id')
319                 if Id is not None:
                        seed_ids.append(Id)
321         unique = list(set(seed_ids))
            # Record the starting nodes for debugging. The busy_timeout prevents a issue where sqlite3 was not waiting long enough. It probably isn't needed but....
323         self.archive.execute("PRAGMA busy_timeout = 30000")
            self.archive.execute('INSERT INTO seed_nodes (name,seed_nodes) VALUES(?,?)',(self.table_name,','.join(unique)))
325         return seed_ids

327     # Returns the graph and writes it to the desktop for testing in Gephi.
        def save_graph_to_graphml(self,xtend=''):
329         edge_raw = multiCol(self.database,'SELECT * FROM '+self.table_name+'Eset where Status="normal"')
            node_raw = multiCol(self.database,'SELECT * FROM '+self.table_name+'Node')
331         G = nx.Graph()
            node_attributes = []
333         for edge in edge_raw:
                if edge[2] == 'normal':
335                 G.add_edge(edge[0],edge[1])
                    G[edge[0]][edge[1]]['status'] = edge[2]
337         archive_id = singleCol(self.archive,'Select Id from '+self.table_name+'Node')
            remaining_nodes = [n for n in G.nodes() if n not in archive_id]
339         print(remaining_nodes)
            print('deleted for being problems')
341         G.remove_nodes_from(remaining_nodes)
            for attr in ['name','faction','br','outfitTag','outfitId','outfitSize','creation_date','login_count','minutes_played','last_login_date']:
343             try:
```

```
                G = self.my_set_thing(G,attr,self.sql_columns_to_dicts
('Node',attr,self.database))
345             except:
                    print('failure on '+attr)
347                 raise
        try:
349             nx.write_graphml(G,'C:\\Users\\-John\\Desktop\\testing
Graphs\\'+self.table_name+xtend+'test.graphml')
        except:
351             nx.write_graphml(G,'C:\\Users\\John\\Desktop\\testing
Graphs\\'+self.table_name+xtend+'test.graphml')
        return G
353

    # Sets all nodes in graph G with a new attribute named attri_name
using values found in a_dict.
355     # Networkx has a function that does this but it always throws
errors if the dict is missing any values in the graph.
    def my_set_thing(self,G,attri_name,a_dict):
        for n in G.nodes():
357             G.node[n][attri_name]=a_dict[n]
        return G
359

361     # Converts a SQL column to a dictionary keys are the character Id
and values are whatever is in the column named col_name
    def sql_columns_to_dicts(self,table,col_name,connection):
363         d = {}
        val = multiCol(connection,'SELECT Id,'+col_name+' FROM '+self.
table_name+table)
365         for i in val:
                d[str(i[0])] = i[1]
367         return d

369     # Erase the tables in the database created by this class, use it
if you have a problem.
    # In theory there is no situation where we would need to remove
archived values.
371     def clear_results(self):
        self.database.execute('DROP TABLE '+self.table_name+'Eset')
373         self.database.execute('DROP TABLE '+self.table_name+'Node')
```

```
            self.database.execute('DROP TABLE '+self.table_name+'History')
375
        # Breaks a list into a list of length n list.
377     def chunks(self,alist,n):
            outList = []
379         for i in range(0,len(alist),n):
                outList.append(alist[i:i+n])
381         return outList

383 # Crawl the playstation 4 servers.
    def run_PS4():
385     faults = []
        # Note that most of these servers have since been merged together.
387     for initials in ['G','Cr','L','S','Ce','P']:

389         x = main_data_crawler(initials)
            print('Now crawling %s' % x.server_name)
391         x.run()

393 # Crawl the 3 PC servers.
    def run_PC():
395     faults = []
        for initials in ['E','C','M']:
397         x = main_data_crawler(initials)
            print('Now crawling %s' % x.server_name)
399         x.run()
```

Utility functions

Various functions and global variables needed as basics for nearly all the code.

9pt

```
import random,math,datetime,json,os,sqlite3

import numpy as np
import statistics as stats
import networkx as nx
import matplotlib.pyplot as plt

from collections import Counter,OrderedDict

# I have two paths to the dropbox. You will want to replace p with whatever path you want to use for your data. There are likely a few other places where you will need to do this.
if os.path.exists('E:\\Dropbox\\PS2Research\\SQL database\\'):
    p = 'E:\\Dropbox'
else:
    p = 'C:\\Users\\John\\Dropbox'

# The connection to the database containing the competition networks.
COMP = sqlite3.connect(p+'\\competition.db')

# The connections to the six databases snapshots.

p = p+'\\PS2Research\\SQL database\\'
# The first three databases connect to the raw data consisting of all nodes active in the last 44 days.
Connery = sqlite3.connect(p+'Connery.db')
Emerald = sqlite3.connect(p+'Emerald.db')
Miller = sqlite3.connect(p+'Miller.db')
# The second three are the subset of nodes active in since the last check.
CW = sqlite3.connect(p+'ConneryWeek.db')
MW = sqlite3.connect(p+'MillerWeek.db')
EW = sqlite3.connect(p+'EmeraldWeek.db')

# Converts these connections into the string of their name, mostly used in graphs and debugging.
nDict = {CW:'CW',EW:'EW',MW:'MW',Connery:'Connery',Emerald:'Emerald',Miller:'Miller',COMP:'COMP'}
```

```
# Takes a SQL connection and a query and returns the first value in every row as a list.
def singleCol(conn,query):
    return [i[0] for i in conn.execute(query).fetchall()]

# Takes a SQL connection and a query and returns each row as a list.
def multiCol(conn,query):
    return [list(i) for i in conn.execute(query).fetchall()]

# Lists the column names in a table_name in con.
def tableFacts(con,table_name):
    columns = [i[1] for i in multiCol(con,'PRAGMA table_info('+table_name+')')]
    print(columns)
    return columns

# Lists the tables in the database.
def tabList(con):
    tables = [i[0] for i in con.execute("SELECT name FROM sqlite_master WHERE type='table' ORDER BY name")]
    print(tables)
    return tables

# Takes a table name and connection and returns the graph including the attributes.
def make_graph(con,table,faction = None):
    edges = multiCol(con,'SELECT Source,Target from '+table+'e')
    # First add all the edges form the eset.
    G = nx.from_edgelist(edges)
    # Then add avatar Ids in order to avoid potentially isolated nodes, if the vertex was already introduced by the edges it wont be duplicated.
    G.add_nodes_from(singleCol(con,'SELECT Id from '+table))
    # If optional faction is provided it only returns the graph consisting of avatars in that faction.
    if type(faction) == str:
        # Building a graph of a subgraph turns it into a normal graph object instead of a subgraph object.
```

```
64          G = nx.Graph(G.subgraph(singleCol(con,"SELECT Id from "+table+
    " "+faction)))
     # Add each attribute one by one.
66      for attr in ['name','faction','status2','kills','br','outfitTag','
    minutes_played','creation_date','login_count','minutes_played','
    last_login_date','outfitSize','status','deaths']:
            try:
68              nx.set_node_attributes(G,attr,dict(multiCol(con,'SELECT Id
    ,'+attr+' from '+table)))
            except sqlite3.OperationalError:
70              print('Error in %s, table %s on attribute %s' % (nDict[con
    ],table,attr))
                raise
72      return G

74# Takes a table name and connection and returns the graph, without
    bothering to assign attributes.
def make_graph_lite(con,table):
76      # It also works on the competition and mega graphs.
        if con == COMP:
78          edges = multiCol(con,'SELECT Source,Target from '+table+'e')
            G = nx.from_edgelist(edges)
80          G.add_nodes_from(singleCol(con,'SELECT name from '+table))
        elif table == 'MegaGraph':
82          edges = [i.split(',') for i in singleCol(con,'SELECT
    SourceTarget from '+table+'Edges')]
            G = nx.from_edgelist(edges)
84          G.add_nodes_from(singleCol(con,'SELECT name from '+table))
        else:
86          edges = multiCol(con,'SELECT Source,Target from '+table+'e')
            G = nx.from_edgelist(edges)
88          G.add_nodes_from(singleCol(con,'SELECT Id from '+table))
        return G
```

Population

9pt

```
from utils_graphing import *

def inactive_count_arg(con):
    # Give a raw count of the number of inactive avatars
    for table in singleCol(con,'SELECT date from date_names'):
        Id = set(singleCol(con,'SELECT Id from '+table+' where status = "dead"'))-set(singleCol(con,'SELECT Id from '+table+' where status2 != "dead"'))
        print(table+','+str(len(Id)))

class population_analysis():
    # Making a class to hold all the methods, cuts down on redundancy.

    def __init__(self,con):
        self.con = con
        self.t = singleCol(con,'SELECT date from date_names')
        self.full_id = self.full_id_everywhere(con)
        self.future = self.future_id_everywhere()

    def full_id_everywhere(self,connection):
        # Creates a dictionary of snapshots to list of Ids for each snapshot in the connection
        id_dict_all = {}
        for table in self.t:
            id_dict_all[table] = singleCol(connection,'SELECT Id FROM '+table)
        return id_dict_all

    def future_id_everywhere(self):
        # The avatar Ids that appear in future snapshots
        future = {}
        for index,cur in enumerate(self.t[:-1]):
            val = set()
            for table in self.t[index+1:]:
                for Id in self.full_id[table]:
                    val.add(Id)
            future[cur] = list(val)
        future[self.t[-1]] = self.full_id[self.t[-1]]
```

```
35          return future

37      def full_id_edges(self,connection):
            # Creates a dictionary of snapshots to edges
39          all_edges = {}
            for table in self.t:
41              all_edges[table] = multiCol(connection,'SELECT Source,
    Target FROM '+table+'e')
            return all_edges
43
def node_differences(con):
45      # How many avatars were added (returning from inactivity or new)
    this snapshot?
      # How many avatars were removed (inactive or dead) this snapshot?
47
      P = population_analysis(con).full_id
49      t = singleCol(con,'SELECT date from date_names')
      table_it = iter(t)
51      print(','+nDict[con])
      print('T1 T2,N(T1)-N(T2),N(T2)-N(T1)')
53      # Use sets to do this quickly and easily.
      for t_index in range(len(t)-1):
55          T1_N = set(P[t[t_index]])
            T2_N = set(P[t[t_index+1]])
57          print(t[t_index]+' '+t[t_index+1]+','+str(len(T1_N-T2_N))+','+
    str(len(T2_N-T1_N)))

59  def run_all_node_diff():
      #Runs the node differences method for each database used for the
    population differences graph
61      for c in [CW,Connery,EW,Emerald,MW,Miller]:
            node_differences(c)
63
def avatar_and_friendship_counts(con):
65      # Find the number of avatars and friendships for each snapshot in
    a database uses make_graph_lite to deal with potential database
    problems like repeated edges

67      record = []
```

```
    for table in singleCol(con,'SELECT date from date_names'):
69      G = make_graph_lite(table)
        record.append('%s,%s,%s' % (table,len(G.nodes()),len(G.edges()
    )))
71  return record

73 def population_graph():
    # outputs the results of avatar_and_friendship_counts for each
   database, used for the population graph.
75  for c in [CW,Connery,MW,Miller,EW,Emerald]:
        print(nDict[c])
77      print('Date,Nodes,Edges')
        print(avatar_and_friendship_counts(c))
79
   def density(con):
81
    # Density of each snapshot.
83
    print(nDict[con])
85  print('Date,Density')
    for table in singleCol(con,'SELECT date from date_names'):
87      G = make_graph_lite(con,table)
        print(table+','+str(nx.density(G)))
89
   def run_all_density():
91  # outputs the results of density for each database, used for the
   density graph.
    for c in [CW,EW,MW,Connery,Emerald,Miller]:
93      density(c)

95 def clustering_coefficent(con):
    # clustering coefficient of each graph
97  print('Date,Average Clustering Coefficient.')
    for table in singleCol(con,'SELECT date from date_names'):
99      G = make_graph_lite(con,table)
        print(table+','+str(nx.average_clustering(G)))
101
   def run_all_clustering():
```

```
103     # outputs the results of clustering_coefficent for each database,
        used for the clustering_coefficent graph.
        for c in [CW,EW,MW,Connery,Emerald,Miller]:
105         clustering_coefficent(c)

107 def List_dead_nodes(con,opp=0):
        # Finds all dead avatars
109
        P = population_analysis(con)
111     print(nDict[con])
        for table in P.t:
113         gone = set(P.full_id[table])-set(P.future[table])
            print(table+','+str(len(gone)))
115         if opp == 1:
                return gone
117
    def List_new_dead_nodes(con,opp=0):
119     # Finds all avatars that are both new and dead

121     P = population_analysis(con)
        print(nDict[con])
123     for table in P.t:
            gone = set(P.full_id[table])-set(P.future[table])
125         new = set(new_avatars_by_snapshot(con,table))
            #set intersection symbol is &.
127         gone = gone&new
            print(table+','+str(len(gone)))
129         if opp == 1:
                return gone
131
    def run_dead_all_check():
133     for c in [CW,EW,MW,Connery,Emerald,Miller]:
            print(nDict[c]+' Abandoned avatars')
135         List_dead_nodes(c)

137 def run_new_dead_all_check():
        for c in [CW,EW,MW,Connery,Emerald,Miller]:
139         print(nDict[c]+' Immediately abandoned avatars')
            List_new_dead_nodes(c)
```

```
141
def new_avatars_run_all():
143     for c in [CW,EW,MW,Connery,Emerald,Miller]:
            print(nDict[c]+' new avatars')
145         for table in singleCol(c,'SELECT date from date_names'):
                print(table+','+str(len(new_avatars_by_snapshot(c,table)))
    )
147
def new_avatars_by_snapshot(con,table):
149     # Returns all avatars with a creation date > the highest value
    found in the previous tables last login date.
     # Resulting in a list of only those avatars who were created this
    snapshot.
151
     t = singleCol(con,'SELECT date from date_names')
153     # Get the previous last tables cut off value.
     prev_t_index = t.index(table)-1
155     # For the first week just say 7 days in the past.
     if prev_t_index == -1:
157         table_max = singleCol(con,'SELECT max_login_date from
    date_names where date = "%s"' % table)[0]-604800
     else:
159         last = prev_t_index
            table_max = singleCol(con,'SELECT max_login_date from
    date_names where date = "%s"' % t[last])[0]
161     new = singleCol(con,'SELECT Id from %s where creation_date > %s' %
     (table,str(table_max)))
     return new
163
def confirmed_dead(con,P,table,opp = False):
165     # Takes a database and the corresponding population analysis
    object and a table.
     # Returns the dead avatars from the table. opp toggles printing
    behavior.
167
     gone = set(P.full_id[table])-set(P.future[table])
169     if opp:
            print(table+','+str(len(gone)))
171     return gone
```

```
173 def confirmed_new_dead(con,P,table,opp = False):
    # Takes a database and the corresponding population analysis
    object and a table.
175     # Returns the new_dead avatars from the table. opp toggles
    printing behavior.

177     gone = set(P.full_id[table])-set(P.future[table])
        new = set(new_avatars_by_snapshot(con,table))
179     new_dead = gone&new
        if opp:
181         print(table+','+str(len(new_dead)))
        return new_dead
183
    def add_status3(con):
185     # Add status3 the column that will store weather a particular
    avatar is new dead new_dead or common (other) in our database
        # Replaces status which is flatly wrong and status2 which tracks
    inactive vs active.
187     # This is the value we use pretty much everywhere in our analysis.

189     P = population_analysis(con)
        for table in singleCol(con,'SELECT date from date_names'):
191     #This adds the correct status3 column.
            if 'status3' not in str(tableFacts(con,table)):
193             con.execute('ALTER TABLE '+table+' ADD COLUMN status3 TEXT
    ')
                new,dead,new_dead = new_avatars_by_snapshot(con,table),
    confirmed_dead(con,P,table),confirmed_new_dead(con,P,table)
195             print(table+' new '+str(len(new))+' dead '+str(len(dead)))
                status_dict = {}
197             #We use all nodes from the table just to make sure that
    all of the rows get accounted for.
                all_nodes = singleCol(con,'SELECT Id from '+table)
199             for i in all_nodes:
                    status_dict[i]='common'
201             for i in dead:
                    status_dict[i]='dead'
203             for i in new:
```

```
                status_dict[i]='new'
            for i in new_dead:
                status_dict[i]='new_dead'
            #Update status3
            for i in all_nodes:
                con.execute('UPDATE '+table+' SET status3 = ? where Id
    = ?',(status_dict[i],i))
            con.commit()
```

Avatar removal analysis.

The mega graph is stored in the SQL database with the node attributes in MegaGraph and edge attributes in the MegaGraphEdges table.

Node attributes

Uptime

The number of snapshots a node appears in.

Downtime

The number of snapshots a node does not appear in after its formation.

triads closed

The number of times friends of a node themselves became friends.

new type

The way the node was added and if it was immediately abandoned or not.

dead type

If the node is classified as dead at the end of our dataset.

Edge attributes

SourceTarget

The two avatar ids the edge connects, this is stored as a single string

due to primary key constraints.

formation

The snapshot the edge first appeared in.

dissolution

The snapshot the edge dissolved in if any.

triads closed

The number of open triads closed by the addition of this edge. ie the mutual friends of the endpoint of a edge.

statusPlus

The type of nodes that the edge connects, for example a edge formed between two new nodes is New_adj,New_adj while a edge between a immediately abandoned avatar and a existing avatar is ND," any edges that connect pairs of existing active nodes that are not new are called normal.

statusMinus

The fate of the endpoints of a edge, if one edge dies then this would be D,"" both dying in the same snapshot would be D,D and so on.

This code is use to construct the mega graphs for analysis of node removal.

9pt

```
# Builds the graph of all edges, and records changes.

from utils_graphing import *

class population_analysis():
    # Mostly uses information for the next class which I used to actually build the graph.
    def __init__(self,con):
        self.con = con
        self.t = singleCol(con,'SELECT date from date_names')
        self.edge_status = {}

    # Gets all Ids from the connection.
    def full_id_everywhere(self):
        id_dict_all = {}
        for table in self.t:
            id_dict_all[table] = singleCol(self.con,'SELECT Id FROM '+table)
        return id_dict_all

    # Sorted edges for every edge in all snapshots.
    def edge_list_everywhere(self):
        edge_id = {}
        for table in self.t:
            edge_id[table] = [sorted(i) for i in multiCol(self.con,'SELECT Source,Target from '+table+'e')]
        return edge_id

    # Creates a record each edge when if a edge is seen in the ith snapshot then ith list element is 1 otherwise 0.
    def edge_activity_history(self):
        edge_record = {}
        edge_dict = self.edge_list_everywhere()
        for i,table in enumerate(self.t):
            for e in edge_dict[table]:
                k = e[0]+','+e[1]
                try:
                    edge_record[k][i] = 1
```

```
                except KeyError:
                    edge_record[k] = [0]*len(self.t)
                    edge_record[k][i] = 1
        return edge_record

    # Working activity history that is FAST!
    def construct_inactivity_history(self):
        Ids = self.full_id_everywhere()
        inactivity_history = {}
        for date_index,date in enumerate(self.t):
            for Id in Ids[date]:
                try:
                    inactivity_history[Id][date] = 1
                except KeyError:
                    inactivity_history[Id] = OrderedDict(list(zip(self.t,[-1]*date_index+[0]*len(self.t))))
                    inactivity_history[Id][date] = 1
        return inactivity_history

    # When each new, new_dead and dead node was created.
    def classify_nodes(self):
        node_new,node_dead = {},{}

        for i,table in enumerate(self.t):
            for Id in singleCol(self.con,'SELECT Id from '+table+' where status3 = "new_dead"'):
                node_new[Id] = ('ND',i)
            for Id in singleCol(self.con,'SELECT Id from '+table+' where status3 = "new"'):
                node_new[Id] = ('N',i)
            for Id in singleCol(self.con,'SELECT Id from '+table+' where status3 = "dead"'):
                node_dead[Id] = ('D',i)
        return node_new,node_dead

    # Classify edges by how they were created.
    def classify_edge(self):
        edge_status,edge_status_d = {},{}
        for table in self.t:
```

```
70              G = make_graph_lite(self.con,table)
                # Get the edges incident to each status.
72              N_edges = self.incident_edges(G,table,'new')
                ND_edges = self.incident_edges(G,table,'new_dead')
74              D_edges = self.incident_edges(G,table,'dead')

76              # First loop finds those edges attached to new and new
    dead characters.
                for elist,label in zip([N_edges,ND_edges],['New_adj','
    ND_adj']):
78                  for e in elist:
                        k = ','.join(e)
80                      try:
                            edge_status[k] = edge_status[k]+','+label
82                      except:
                            edge_status[k] = label
84              # Second loop finds those edges attached to dead
    characters.
                for e in D_edges:
86                  k = ','.join(e)
                    edge_status_d[k] = 'D_adj'
88          self.edge_status = edge_status
            return edge_status,edge_status_d
90
     # Returns all edges adjacent to a specific status of node.
92      def incident_edges(self,G,table,status):
            output = []
94          for nodes in singleCol(self.con,'SELECT Id FROM '+table+'
    WHERE status3 == "%s"' % status):
                for adj in G[nodes]:
96                  output.append(sorted([nodes,adj]))
            return output
98


100


102
# Constructs the mega graph from one of the database connections.
104 class mega_graph_constructor():
```

```
    # Restrict tables to the first cutoff tables
106 def __init__(self,con,cuttoff = None):
        self.P = population_analysis(con)
108     if cuttoff is not None:
            self.P.t = self.P.t[:cuttoff]
110     self.G = nx.Graph()
        self.con = con
112     self.preliminaries()
        self.run()
114
    # Use functions from population analysis to get the data we need
    to build our graph, takes a bit to run usually.
116 def preliminaries(self):
        self.edge_hist = self.P.edge_activity_history()
118     self.node_hist = self.P.construct_inactivity_history()
        self.edge_status,self.edge_status_d = self.P.classify_edge()
120     self.new_dict,self.dead_dict = self.P.classify_nodes()
        print('Preliminary values gathered.')
122
    def run(self):
124     self.add_edges()
        self.friendship_labels()
126     self.add_nodes()
        self.character_labels()
128     self.add_names_factions()
        self.first_formation_of_an_edge()
130     self.edge_breaking()


132
        self.triadic_analysis()
134     print('Complete')

136 # Add the edges to the graph.
    def add_edges(self):
138     for eset in self.P.edge_list_everywhere().values():
            self.G.add_edges_from(eset)
140
    # Add nodes to the graph, thus avoiding problems with isolated
    characters. And labels each edge with type new new_dead and dead.
```

```
142     def add_nodes(self):
            node_count = len(self.G.nodes())
144         self.G.add_nodes_from(list(self.new_dict.keys())+list(self.
    dead_dict.keys()))
            print('isolated nodes found = '+str(len(self.G.nodes())-
    node_count))
146
     # Adds the avatar names and factions along with the original
    server if applicable.
148     def add_names_factions(self):
            name_dict,faction_dict = {},{}
150         for table in self.P.t:
                for Id,name,fac in multiCol(self.con,'SELECT Id,name,
    faction from '+table):
152                 # Find names.
                    if name != '' or None:
154                     name_dict[Id] = name
                    # Find factions.
156                 if fac in ['NC','TR','VS']:
                        faction_dict[Id] = fac
158
            # In the case that we are looking at the mega graph of emerald
     assign its server of origin to each Id.
160         if nDict[self.con] in ['EW','Emerald']:
                # Get the Ids from the 3 snapshots of waterson and
    mattherson.
162             WorM_dict = {}
                for server,label in zip(['waterson','mattherson'],['W','M'
    ]):
164                 for date in ['May18','June15','June23']:
                        for Id in singleCol(Emerald,'SELECT Id FROM %s' %
    server+date):
166                         WorM_dict[Id] = label
                # Assign its server of origin.
168             for Id,fac in faction_dict.items():
                    faction_dict[Id] = WorM_dict.get(Id,'N')+fac
170
            for Id in self.G.nodes():
172             self.G.node[Id]['name'] = name_dict.get(Id,'error')
```

```
            self.G.node[Id]['faction'] = faction_dict.get(Id,'error')

    # Adds the Ntime, Ntype, Dtime and Dtype attributes to each
    character Id in the mega graph.
    def character_labels(self):
        for k,v in self.new_dict.items():
            self.G.node[k]['Ntype'],self.G.node[k]['Ntime'] = v
        for k,v in self.dead_dict.items():
            self.G.node[k]['Dtype'],self.G.node[k]['Dtime'] = v

    # Labels each edge adjacent to a new or new_dead Id and all edges
    adj to a dead node. It takes into account the status of both
    endpoints.
    def friendship_labels(self):
        print(list(self.edge_status.items())[:2])
        for k,v in list(self.edge_status.items()):
            e1,e2 = k.split(',')
            self.G.edge[e1][e2]['formation'] = self.edge_hist[k].index
    (1)
            self.G.edge[e1][e2]['status+'] = v
            if v != self.G.edge[e1][e2]['status+']:
                print('Error6')

        for k,v in list(self.edge_status_d.items()):
            h = self.edge_hist[k]
            e1,e2 = k.split(',')
            self.G.edge[e1][e2]['dissolution'] = (len(h) - 1) - h
    [::-1].index(1)
            self.G.edge[e1][e2]['status-'] = v

    def first_formation_of_an_edge(self):

        # My earlier node labels only give formation/dissolution
    indices to the edges that are connected to new or dead characters.
    Neglecting the possibility of formation between existing edges.

        # history of nodes flattens the ordered dictionary we stored
    our node history as into a list of zeros and 1s in the same order.
        history_of_nodes = {}
```

```
204
        for k,v in self.node_hist.items():
206             history_of_nodes[k] = list([v[i] for i in self.P.t])

208         for edge in self.G.edges():
                e1,e2 = edge
210             #Find the first appearance of both endpoints.
                A1 = history_of_nodes[e1].index(1)
212             A2 = history_of_nodes[e2].index(1)
                #Then find the first appearance of the edge that will
    eventually join them.
214             EA = self.edge_hist[','.join(sorted([e1,e2]))].index(1)
                #If both nodes appeared before the first appearance of the
     edge that will eventually join them we can say that edge has a
    first appearance.
216             if A1 < EA and A2 < EA:
                    self.G.edge[e1][e2]['firstAppeared'] = EA
218                 try:
                        self.G.edge[e1][e2]['status+'] = self.G.edge[e1][
    e2]['status+']+',normal'
220                 except:
                        self.G.edge[e1][e2]['status+'] = 'normal'
222
     def edge_breaking(self):
224
        # Looks at the changes in friend relations tries to find
    examples of edges that get removed and adds appropriate labels to
    the mega graph.
226     # Also sees if we have off again on again friendships, and
    records the final state of each edge plus is for existent and minus
    for removed.

228     results = {}
        error_nodes = []
230     for edge in self.G.edges():
            #This section is kind of a mess because I originally had
    it running on only edges adjacent to new characters instead of on
    all edges. Works fine though.
232         k = ','.join(sorted(edge))
```

```
            v = list(self.edge_hist[k])
234         starting_index = v.index(1)
            loc = {'+':0,'-':0,'off_on':'','end_state':''}
236         e1,e2 = k.split(',')
            try:
238         # Active vs inactive history for both endpoints/
                hist_e1,hist_e2 = list(self.node_hist[e1].values()),
    list(self.node_hist[e2].values())
240         except:
                error_nodes.append([e1,e2])
242             print(len(error_nodes))
                continue
244         off_on_again = ''
            for i,list_value in enumerate(v):
246             if list_value == 1:
                    loc['+'] += 1
248                 off_on_again = off_on_again +'+'
                    loc['end_state'] = '+'
250             elif list_value == 0:
                    loc['-'] += 1
252                 off_on_again = off_on_again +'-'
                    loc['end_state'] = '-'
254         #Removes repeated signs, not the most elegant.
            while '++' in off_on_again:
256             off_on_again = off_on_again.replace('++','+')
            while '--' in off_on_again:
258             off_on_again = off_on_again.replace('--','-')
            starting_index = v.index(1)
260         self.G.edge[e1][e2]['formation'] = v.index(1)
            self.G.edge[e1][e2]['+'] = loc['+']
262         self.G.edge[e1][e2]['-'] = loc['-']
            self.G.edge[e1][e2]['state'] = loc['end_state']
264         if len(off_on_again) >= 2:
                self.G.edge[e1][e2]['on_off'] = off_on_again
266

 def triadic_analysis(self):
268     # Adds attributes to both nodes and edges each edge that is
    created counts how many triangles (mutual friends of its endpoints)
    were created by its existence.
```

```
        # Each node records the total number of times it has been one
    of the mutual friends.
270
        formed_edge_dict = nx.get_edge_attributes(self.G,'
    firstAppeared')
272
        Loc_G = nx.Graph()
274     #The number of mutual friends between the endpoints of a edge.
        triads_closed_by_edge = {}
276
        #Every time mutual friends of a character become friends
    increment.
278     triads_closed_by_character = {}
        i = 1
280     for eset in self.P.edge_list_everywhere().values():
            #Time step holds all normal edges formed during this eset.
282         timestep = []
            for k,v in formed_edge_dict.items():
284             if v == i:
                    timestep.append(k)
286
            Loc_G.add_edges_from(eset)
288         #Get the neighborhoods of each
            for edge in timestep:
290             e1,e2 = sorted(edge)
                try:
292                 neighbourhood1,neighbourhood2 = set(Loc_G[e1]),set
    (Loc_G[e2])
                except KeyError:
294                 break
                overlap = neighbourhood1.intersection(neighbourhood2)
296             triads_closed_by_edge[e1+','+e2] = len(overlap)
                for characters in overlap:
298                 try:
                        triads_closed_by_character[characters]+=1
300                 except:
                        triads_closed_by_character[characters] = 1
302         i = i+1
        #Write to the mega graph, as per usual.
```

```
304         for Id in self.G.nodes():
                self.G.node[Id]['triad_closed'] =
    triads_closed_by_character.get(Id,'')
306         for edge in self.G.edges():
                e1,e2 = sorted(edge)
308             self.G.edge[e1][e2]['triad_closed'] =
    triads_closed_by_edge.get(e1+','+e2,'')
            return triads_closed_by_edge,triads_closed_by_character
310

312     def summary(self):
            # Output a summary of some simple factors so I can add them as
     a table to the thesis.
314
            print('Summary '+nDict[self.con])
316         for col in ['status+','status-','state','on_off']:
                val = nx.get_edge_attributes(self.G,col).values()
318             count = Counter(val)
                output = []
320             for i in list(count.keys()):
                    output.append([i,count[i]])
322             print(col)
                print(str(output))
324         for col in ['+','-','formation','dissolution']:
                val = nx.get_edge_attributes(self.G,col).values()
326             count = Counter(val)
                output = []
328             for i in sorted(list(count.keys())):
                    output.append([i,count[i]])
330             print(col)
                print(str(output))
332
    def save_graph(self,path = 'C:\\Users\\John\\Desktop\\
    mega_G_output_test'):
334         # Saves the graph as a graphml file you will need to give it a
     path for your computer.
            nx.write_graphml(self.G,path+'_'+nDict[self.con]+'.graphml')
336         x = nx.get_node_attributes(self.G,'Ntime')
```

```
        with open('C:\\Users\\John\\Desktop\\'+nDict[self.con]+'_repair.csv','a') as f:
338             for k,v in x.items():
                    f.write(str(k)+','+str(v)+'\n')
340
    def save_dead_examining_graph(self,path = 'C:\\Users\\John\\Desktop\\mega_G_output_test'):
342         #The last 6 weeks of data can have inaccurate node death information thus we must cut them off.

344         delete_me = []
            lG = self.G
346         for n in lG.nodes():
                cutoff = len(self.P.t) - 6
348             if lG.node[n]['Ntime'] > cutoff or lG.node[n]['Dtime'] > cutoff:
                    delete_me.append(n)
350         lG.remove_nodes_from(delete_me)

352         nx.write_graphml(lG,path+'_'+nDict[self.con]+'AccDeath.graphml')
            return lG
354
def save_mega_graph(Constructor):
356
    # Save the final mega graph to the database, takes the mega_graph_constructor object.
358
    Connection = Constructor.con
360 G = Constructor.G
    #Create a table for nodes and there attributes
362 Connection.execute('Create table if not exists MegaGraph (Id Primary key,name TEXT,faction TEXT,triads_closed INTEGER,Ntype TEXT, Dtype TEXT,Ntime INTEGER,Dtime INTEGER)')
    #Create a table for edges and all corresponding
364 Connection.execute('Create table if not exists MegaGraphEdges (SourceTarget Primary key,formation INTEGER,dissolution INTEGER, triads_closed INTEGER, statusPlus INTEGER, statusMinus INTEGER, on_off TEXT, plus INTEGER,minus INTEGER,firstAppeared INTEGER)')
```

```
    #Insert the nodes
366 for n in G.nodes():
        attributes = G.node[n]
368     line = n,attributes.get('name',''),attributes.get('faction','')
    ),attributes.get('triad_closed',''),attributes.get('Ntype',''),
    attributes.get('Dtype',''),attributes.get('Ntime',''),attributes.get
    ('Dtime','')
        Connection.execute('INSERT OR REPLACE INTO MegaGraph (Id,name,
    faction,triads_closed,Ntype,Dtype,Ntime,Dtime) values
    (?,?,?,?,?,?,?,?)',line)
370 Connection.commit()
    #Insert the edges.
372 for edge in G.edges():
        e1,e2 = sorted(edge)
374     attributes = G.edge[e1][e2]
        line = e1+','+e2,attributes.get('formation',''),attributes.get
    ('dissolution',''),attributes.get('triad_closed',''),attributes.get(
    'status+',''),attributes.get('status-',''),attributes.get('on_off','
    '),attributes.get('plus',''),attributes.get('minus',''),attributes.
    get('firstAppeared','')
376     Connection.execute('INSERT OR REPLACE INTO MegaGraphEdges (
    SourceTarget,formation,dissolution,triads_closed,statusPlus,
    statusMinus,on_off,plus,minus,firstAppeared) values
    (?,?,?,?,?,?,?,?,?,?)',line)
    Connection.commit()
```

Small world code.

The calculation were done using networkX functions. Both of which use the brute force networkX where $APL = \sum_{s,t \in V(G)} \frac{d(s,t)}{N(N-1)}$ while the diameter of the graph is computed by finding the largest shortest path length $max_{s,t \in V(G)}(d(s,t))$.

9pt

```
# Find the average path length, and diameter of every snapshot..
from utils_graphing import *

#Returns dates from a database.
class component_handler():
    '''Dealing with components, given some graph it can give us the size of each component and can return the largest connected component for use elsewhere.'''
    def __init__(self,Graph):
        self.G = Graph
        #List of all the nodes in each connected component of the graph (undirected) sorted largest component to smallest.
        self.component_list = sorted(nx.connected_components(Graph), key = len, reverse=True)

    def get_giant_component(self):
        '''Removes all nodes that are not part of the giant (largest) component.'''
        Delete_me = []
        for i in self.component_list[1:]:
            Delete_me.extend(i)
        self.G.remove_nodes_from(Delete_me)

    def component_sizes(self):
        '''Returns a list of the size of all components.'''
        return [len(i) for i in self.component_list]

#Average_path_length calculation.  Make sure to not to run this on large graphs of you want to get any other work done.
def calculate_average_path(Graph,TE = False):
    C = component_handler(Graph)
    C.get_giant_component()
    if TE:
        x = nx.average_shortest_path_length(C.G)
        print(str(x))
    else:
        print(str(nx.average_shortest_path_length(C.G))+',')
```

```
#Find average path lenghts
def average_path_length_data(con,faction = 'where faction = TR',R = 0):
    print(nDict[con]+' '+faction+' average shortest path length')
    for t in singleCol(con,'SELECT date from date_names')[R:]:
        print(''"+t)
        Graph = make_graph(con,t)
        G = nx.Graph(Graph.subgraph(singleCol(con,"SELECT Id from "+t+" "+faction)))
        calculate_average_path(G,TE = True)

#Calculation of diameter, Make sure to only use the smallest graph you can find.
def calculate_diameter(Graph,TE = False):
    C = component_handler(Graph)
    C.get_giant_component()
    if TE:
        x = nx.diameter(C.G)
    else:
        print(str(nx.diameter(C.G))+',')
    return x

#Find the diameter, for every snapshot with index >= R
def diameter_data(con,faction = "where faction = TR",R = 0):
    print(nDict[con]+' '+faction+' diameter')
    for t in singleCol(con,'SELECT date from date_names')[R:]:
        w = ''"+t
        Graph = make_graph(con,t)
        G = nx.Graph(Graph.subgraph(singleCol(con,"SELECT Id from "+t+" "+faction)))
        w = w+','+str(calculate_diameter(G,TE = True))
        print(w)
```

Chapter 9

Appendix 2: Additional information.

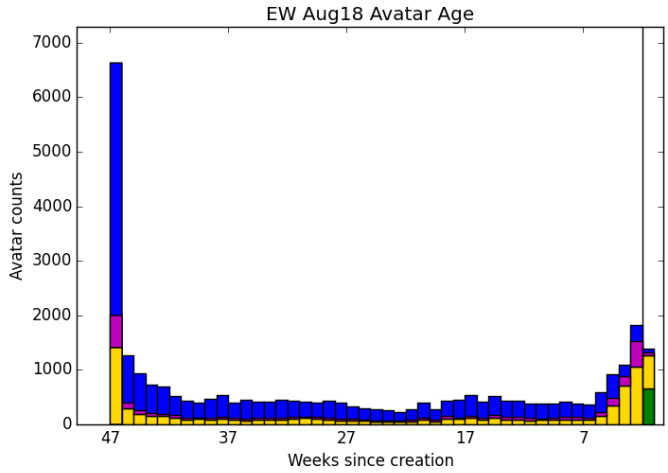
SOE and DBG.

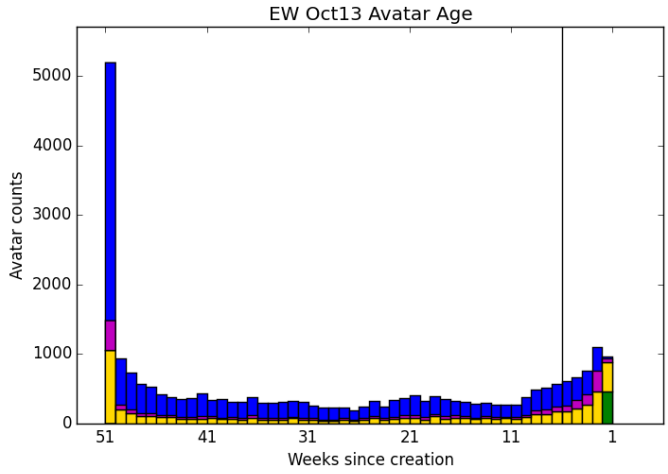
The company that develops PlanetSide 2 was named Sony Online Entertainment (SOE) however they are now Daybreak Games (DBG) after being sold by Sony in the winter of 2015. They have previously produced a number of other notable massively multiplayer online games (MMOG's) such as EverQuest and Star wars Galaxies.

Additional figures

Avatars by age.

The other two servers are shown in Figure 9.1 and Figure 9.2. As before the blue represents overall population. The gold indicates the avatars who reactivated while the magenta and green indicates the avatars who are dead and new respectively. The vertical black line indicates the week we started collecting data.


EW Aug18 Avatar Age
Avatar counts
Weeks since creation
0
1000
2000
3000
4000
5000
6000
7000
47
37
27
17
7


EW Oct13 Avatar Age
Avatar counts
Weeks since creation
0
1000
2000
3000
4000
5000
51
41
31
21
11
1

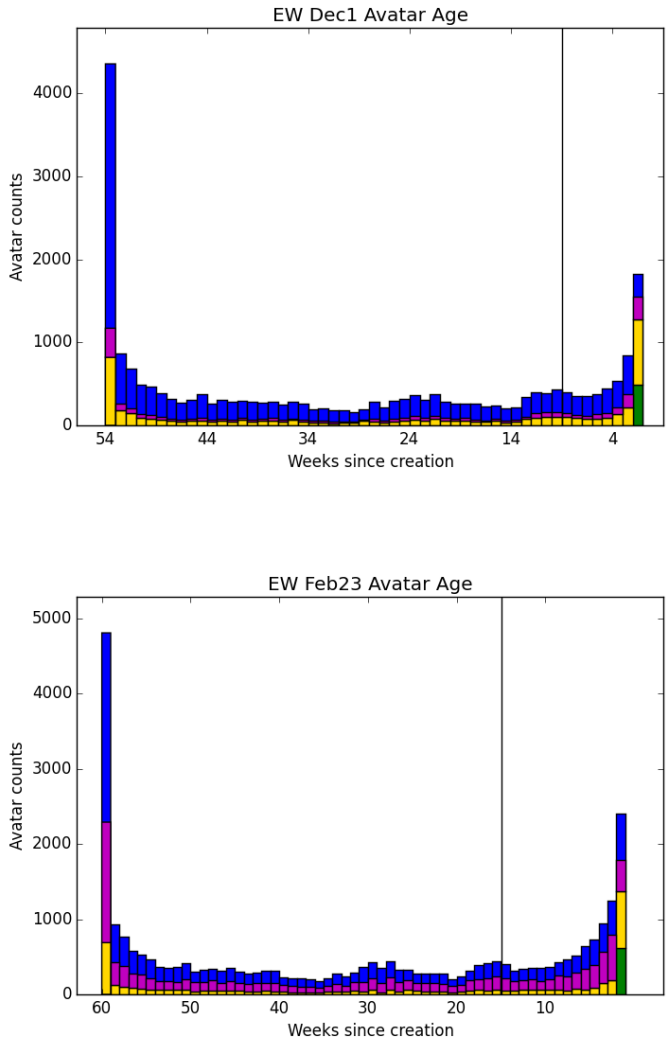


Figure 9.1: Additional avatars by creation date EW.

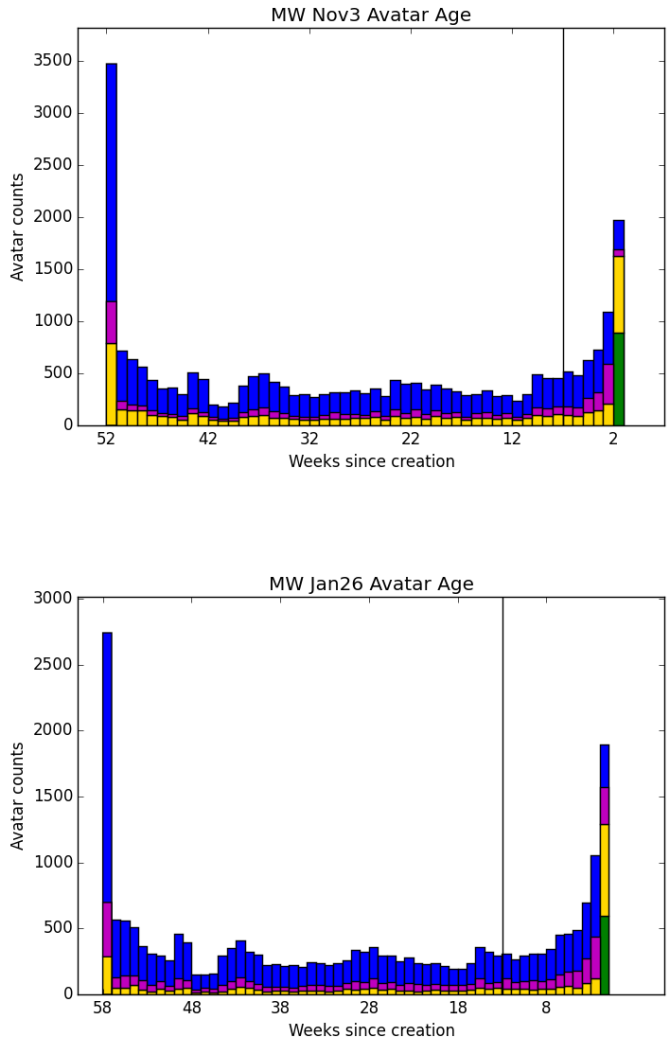
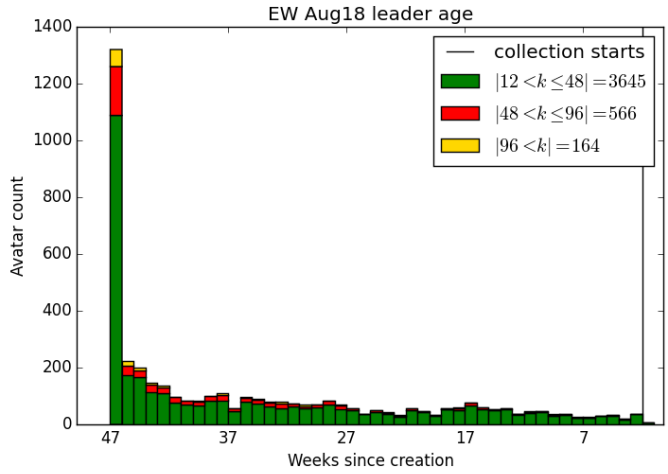
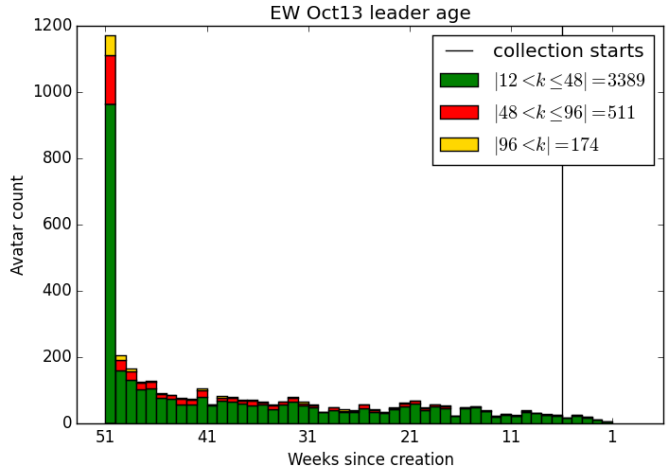


Figure 9.2: Additional avatars by creation date MW.

Hubs by ages

The other two servers are shown in figure 9.3 and figure 9.4. As before gold indicates avatars with degree of 96 or more, while red and green are avatars with degrees of 48 to 96 and 12 to 48.


EW Aug18 leader age
collection starts
$|12 < k \leq 48| = 3645$
$|48 < k \leq 96| = 566$
$|96 < k| = 164$
Avatar count
Weeks since creation
1400
1200
1000
800
600
400
200
0
47
37
27
17
7


EW Oct13 leader age
collection starts
$|12 < k \leq 48| = 3389$
$|48 < k \leq 96| = 511$
$|96 < k| = 174$
Avatar count
Weeks since creation
1200
1000
800
600
400
200
0
51
41
31
21
11
1

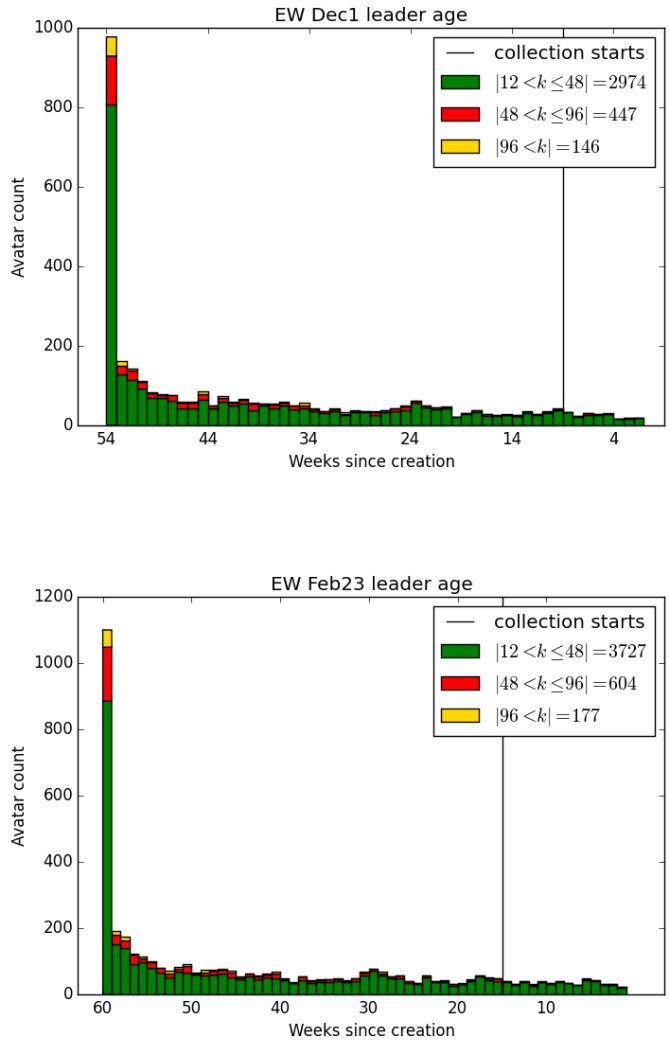


Figure 9.3: High degree avatars by creation date for EW.

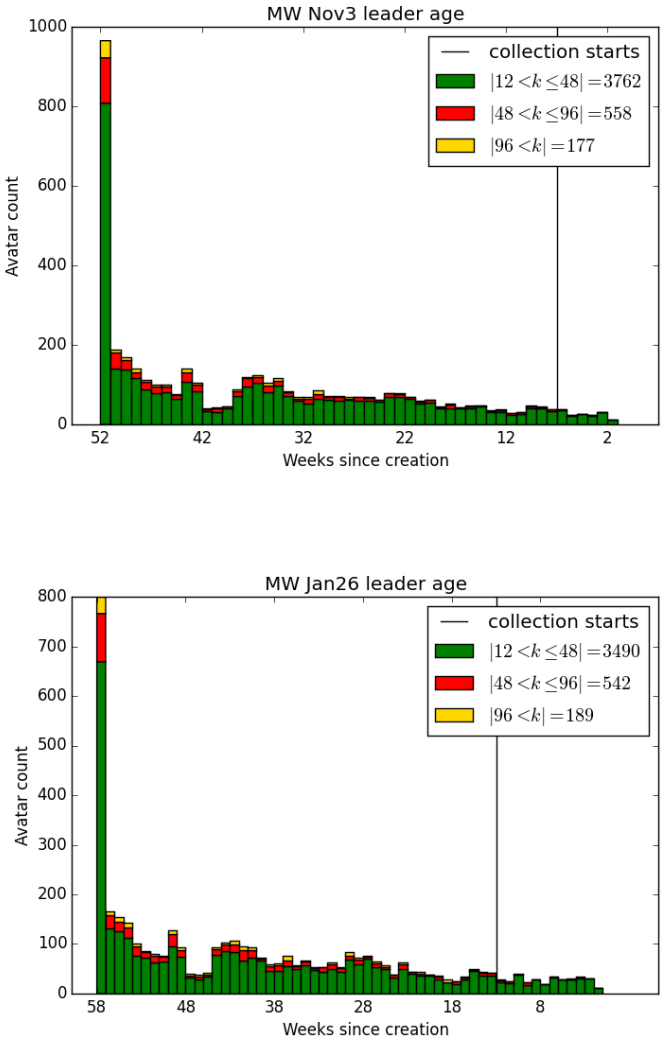


Figure 9.4: High degree avatars by creation date for MW.

Clustering coefficient

Figure 9.5 shows example clustering coefficient distributions from EW and MW.

Outfit size

Figure 9.6 has examples of avatar states by size of their outfit.

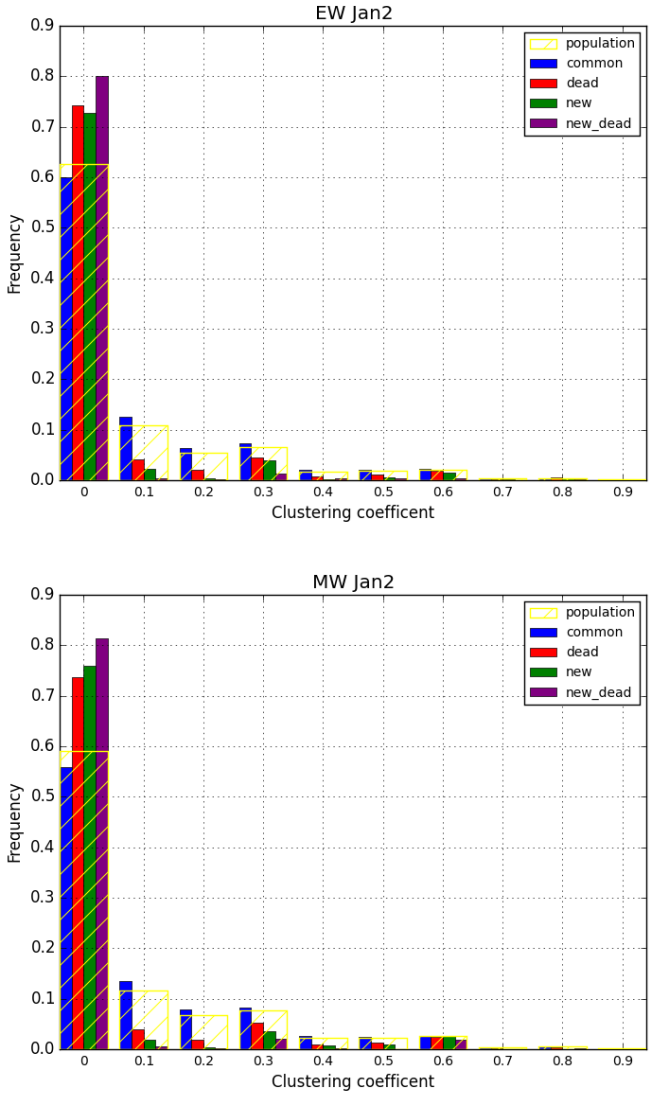


Figure 9.5: Additional clustering coefficient examples.

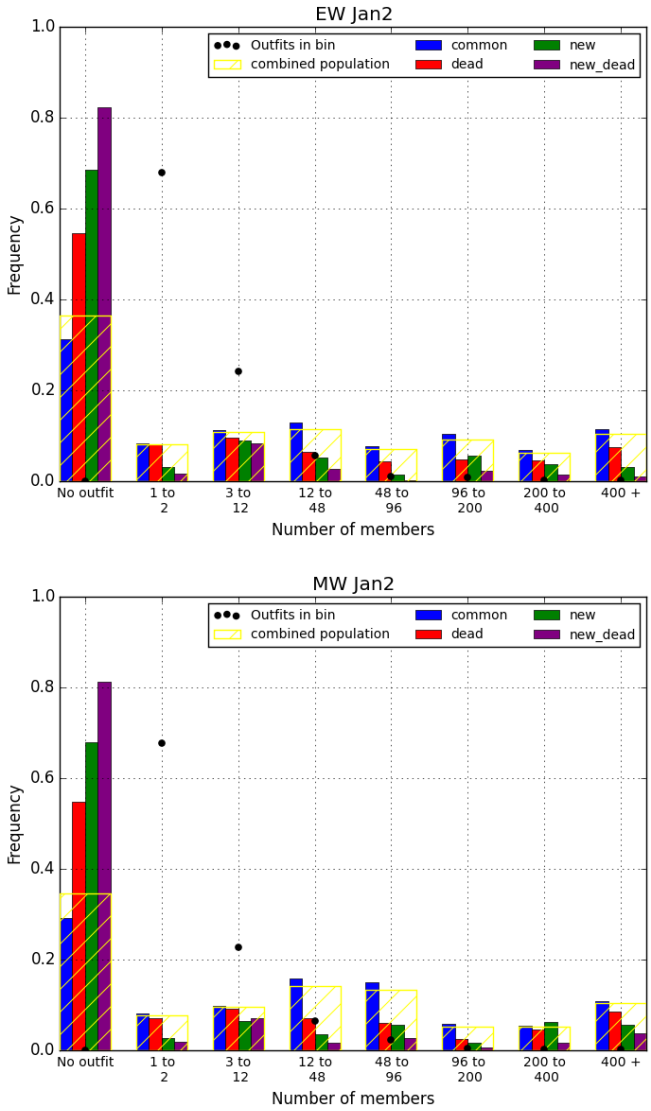


Figure 9.6: Additional avatar states by size of outfit.

Battle rank.

And finally examples of avatar state by battle rank can be found in 9.7 for EW and MW respectively.

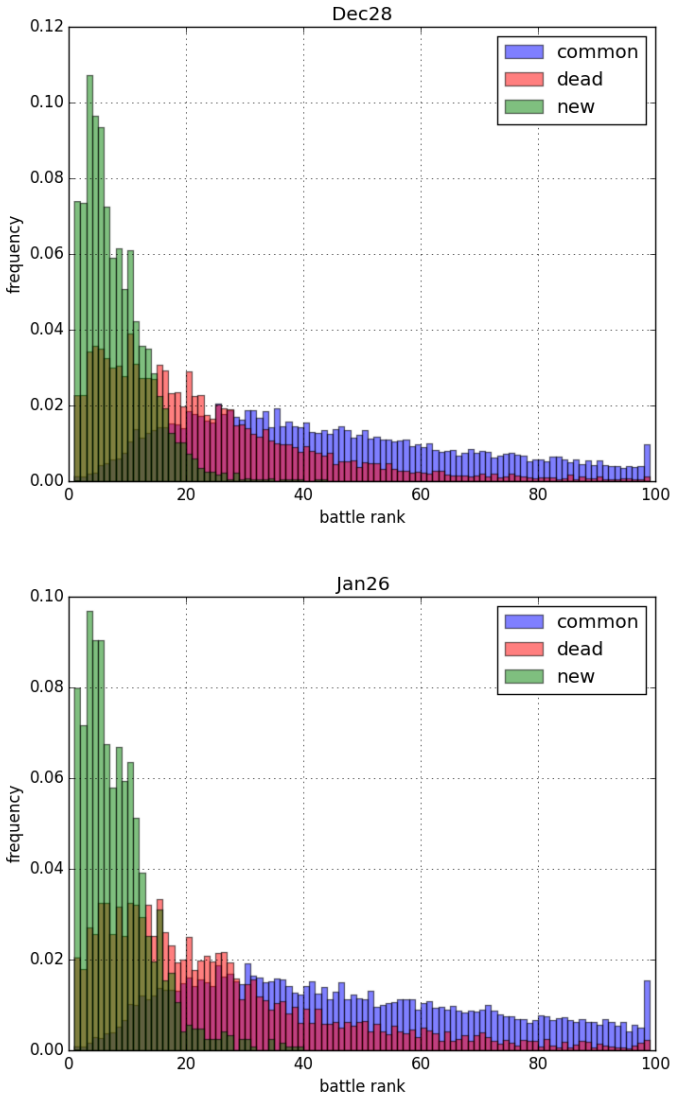


Figure 9.7: Additional examples of avatar state by battle rank.

Gephi visualizations

These visualizations of show more conventional social networks under the force atlas 2 algorithm using the same parameters as the merger figures. Both the Figures 9.8 and 9.9 are from the Stanford large network dataset collection [15].

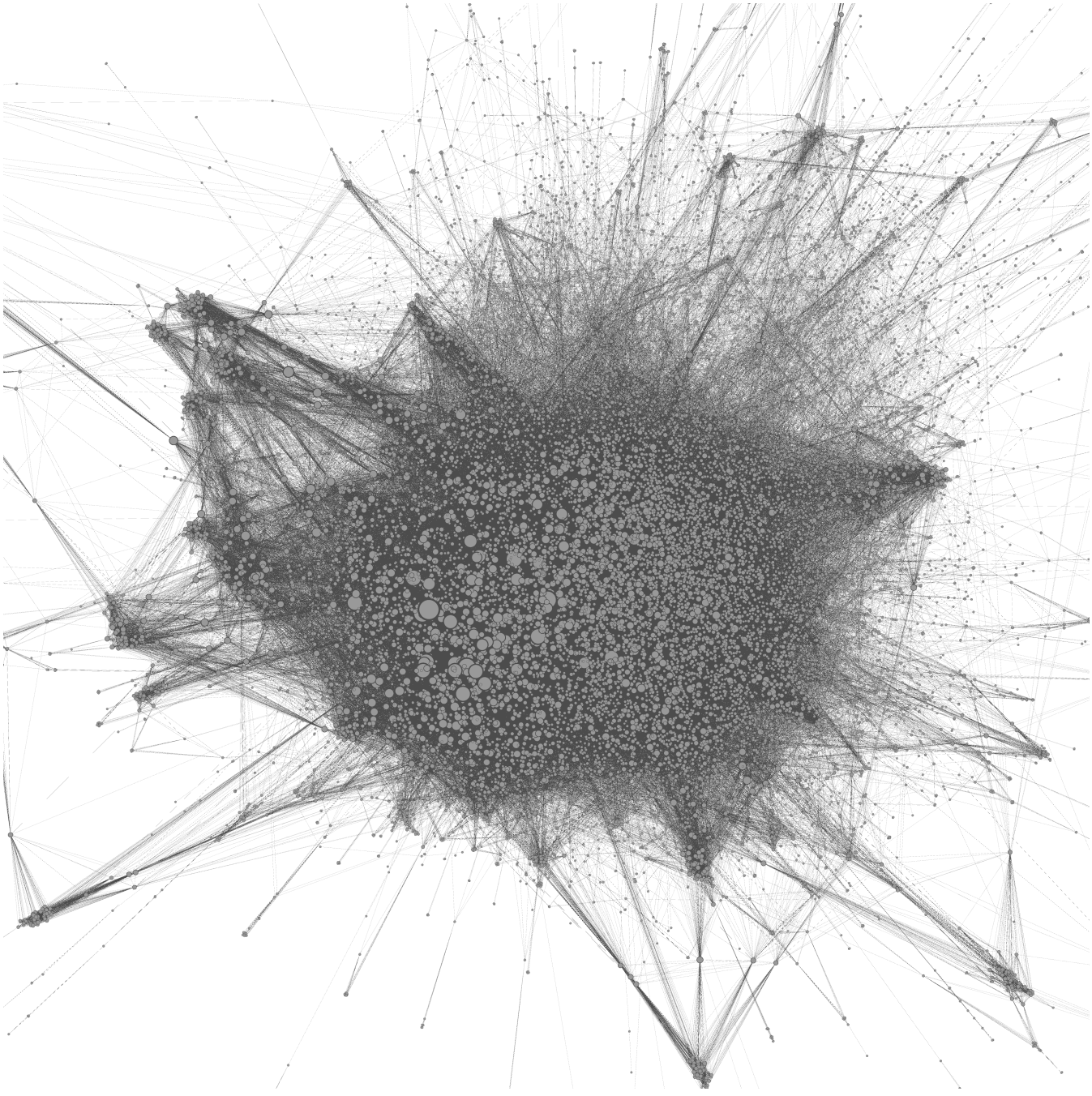


Figure 9.8: The Astrophysics collaboration network.

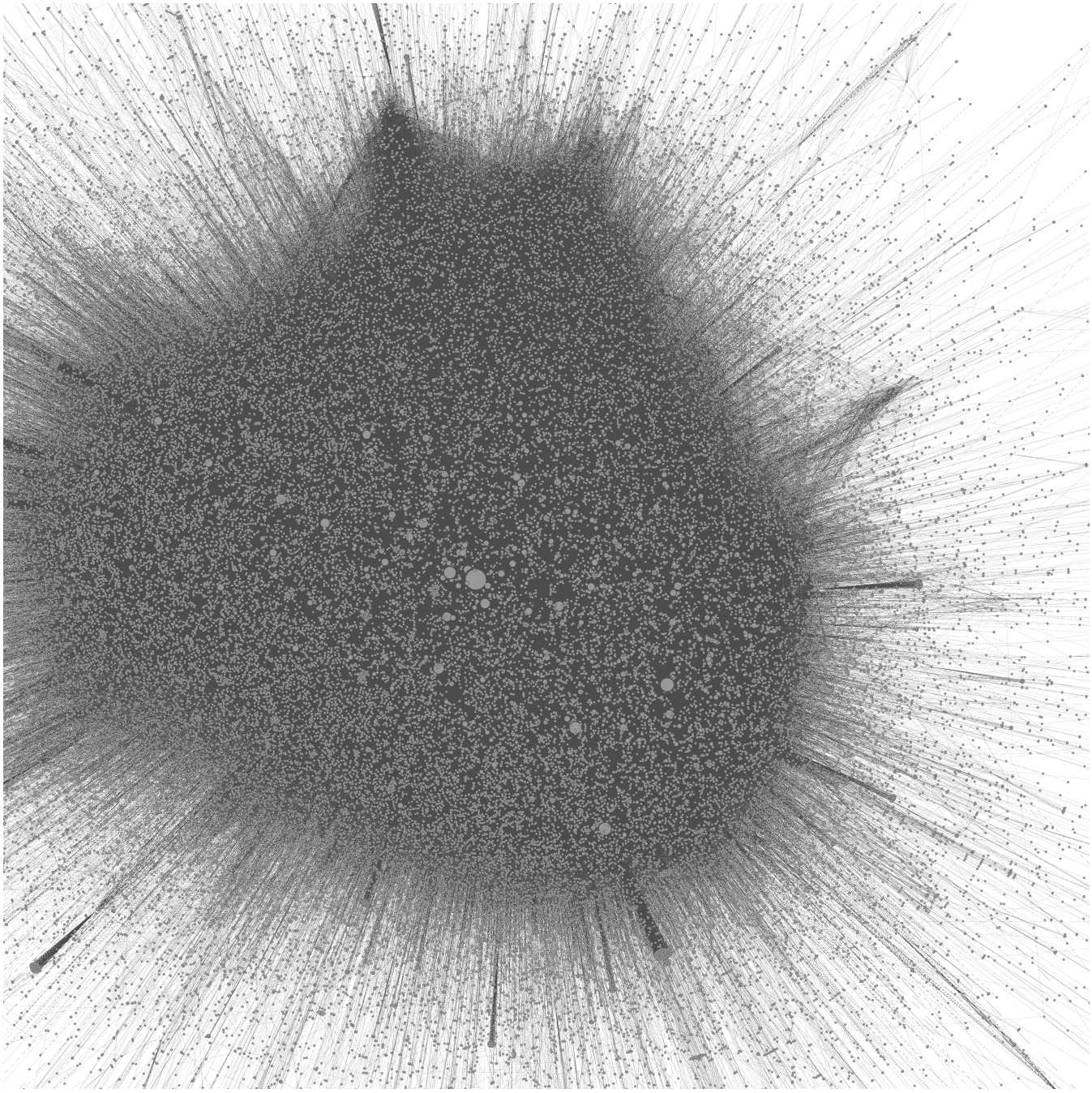


Figure 9.9: The network of who trusts whom on epinions.

Date of PlanetSide 2 snapshots

In addition to the dates in Table 9.1 there are snapshots of Mattherson and Waterson on May 18th, and the 15th and 23rd of June. As well as weekly snapshots of both Emerald and Connery for July and early August, however these early snapshots only contained the Id, name, br, and outfit tag (rather then outfit id). So it is not possible to add these to the week snapshots.

PlanetSide 2 player group structure.

In PlanetSide 2 players organize themselves the squads and platoon system, a single squad consists of a squad leader and up to 11 other players while a platoon consists of 2 to 4 squads. The full platoon of 48 players is the largest unit supported in the game.

Power law fits by snapshot.

Snapshots			
Week	Connery	Emerald	Miller
1	11-Aug-14	11-Aug-14	
2	18-Aug-14	18-Aug-14	
3		25-Aug-14	
4	1-Sep-14	1-Sep-14	
5	9-Sep-14	9-Sep-14	
6	15-Sep-14	15-Sep-14	
7	22-Sep-14	22-Sep-14	
8	1-Oct-14	1-Oct-14	
9	7-Oct-14	7-Oct-14	
10	13-Oct-14	13-Oct-14	
11	20-Oct-14	20-Oct-14	23-Oct-14
12	27-Oct-14	27-Oct-14	
13	3-Nov-14	3-Nov-14	3-Nov-14
14	11-Nov-14	12-Nov-14	12-Nov-14
15	17-Nov-14	17-Nov-14	17-Nov-14
16	26-Nov-14	26-Nov-14	26-Nov-14
17	4-Dec-14	1-Dec-14	3-Dec-14
18	8-Dec-14		8-Dec-14
19	18-Dec-14	17-Dec-14	17-Dec-14
20	23-Dec-14		
21	28-Dec-14	28-Dec-14	29-Dec-14
22	2-Jan-15	2-Jan-15	2-Jan-15
23	10-Jan-15	10-Jan-15	10-Jan-15
24	18-Jan-15	18-Jan-15	18-Jan-15
25	26-Jan-15	26-Jan-15	26-Jan-15
26	31-Jan-15	31-Jan-15	31-Jan-15
27	8-Feb-15	8-Feb-15	8-Feb-15
28	15-Feb-15	15-Feb-15	15-Feb-15
29	23-Feb-15	23-Feb-15	23-Feb-15
30	1-Mar-15	1-Mar-15	1-Mar-15

Table 9.1: The date each of snapshot.

CW				Log likelihood		
Date	xmin	D	α	$x^{-\alpha}$ vs $e^{-\lambda x}$ (R,p)	$x^{-\alpha}$ vs $x^{-\alpha}e^{-\lambda x}$ (R,p)	n tail n tail
Aug11	103	0.038	2.223	(3.3872,0.0007)	(-0.5143,0.6827)	141
Aug18	89	0.033	2.164	(3.2755,0.0011)	(-0.7673,0.471)	77
Sept1	111	0.031	2.153	(3.14,0.0017)	(-0.7609,0.4534)	191
Sept9	94	0.035	2.185	(3.3543,0.0008)	(-0.7398,0.4945)	185
Sept22	98	0.033	2.154	(3.12,0.0018)	(-0.7889,0.4433)	162
Oct1	92	0.026	2.172	(3.0401,0.0024)	(-0.7817,0.434)	60
Oct13	88	0.051	2.156	(2.934,0.0033)	(-1.4471,0.101)	118
Oct20	90	0.039	2.234	(2.9993,0.0027)	(-0.7438,0.4758)	101
Oct27	94	0.035	2.246	(3.2144,0.0013)	(-0.6852,0.5259)	137
Nov3	103	0.035	2.287	(2.955,0.0031)	(-0.6093,0.5595)	111
Nov11	105	0.038	2.221	(2.8186,0.0048)	(-0.7823,0.4158)	162
Nov17	87	0.034	2.171	(3.1989,0.0014)	(-0.8077,0.4285)	95
Nov26	96	0.043	2.164	(3.3046,0.001)	(-0.8562,0.3995)	151
Dec4	95	0.038	2.179	(3.3841,0.0007)	(-0.8003,0.4521)	183
Dec8	83	0.043	2.155	(3.4864,0.0005)	(-0.9378,0.3666)	98
Dec18	100	0.035	2.169	(3.2903,0.001)	(-0.9163,0.3587)	264
Dec23	99	0.037	2.183	(3.3151,0.0009)	(-0.8621,0.3988)	149
Dec28	94	0.051	2.155	(3.2487,0.0012)	(-0.9338,0.3659)	98
Jan2	84	0.044	2.147	(3.2801,0.001)	(-0.8923,0.3835)	163
Jan10	85	0.055	2.148	(3.3626,0.0008)	(-0.8198,0.4521)	108
Jan18	100	0.050	2.216	(3.318,0.0009)	(-0.6764,0.56)	26
Jan26	100	0.041	2.221	(3.0219,0.0025)	(-0.6546,0.5452)	134
Jan31	97	0.047	2.272	(3.1924,0.0014)	(-0.5088,0.6893)	138
Feb8	106	0.040	2.286	(3.2543,0.0011)	(-0.5685,0.6356)	27
Feb15	112	0.048	2.287	(2.9358,0.0033)	(-0.6281,0.5618)	173
Feb23	120	0.053	2.277	(3.1333,0.0017)	(-0.5765,0.6146)	152
Mar1	104	0.038	2.247	(3.0043,0.0027)	(-0.643,0.5571)	17
Mean	117.964	0.052	2.237	(3.0933,0.0043)	(-0.4050,0.7787)	163.786
SD	46.713	0.008	0.074	(0.4346,0.0127)	(0.1288,0.1234)	38.686

Table 9.2: Optimal power law fits for each Connery snapshot.

EW				Log likelihood		
Date	xmin	D	α	$x^{-\alpha}$ vs $e^{-\lambda x}$ (R,p)	$x^{-\alpha}$ vs $x^{-\alpha}e^{-\lambda x}$ (R,p)	n tail
Aug11	100	0.046	2.277	(3.6017,0.0003)	(-0.5566,0.6855)	190
Aug18	108	0.048	2.214	(3.4755,0.0005)	(-0.3781,0.8218)	117
Aug25	105	0.036	2.200	(3.4382,0.0006)	(-0.4197,0.789)	156
Sept1	112	0.050	2.255	(3.5277,0.0004)	(-0.298,0.8766)	208
Sept9	106	0.044	2.225	(3.4852,0.0005)	(-0.4163,0.795)	254
Sept15	100	0.040	2.209	(3.5201,0.0004)	(-0.4074,0.8055)	198
Sept22	114	0.057	2.263	(3.6545,0.0003)	(-0.2733,0.8926)	151
Oct1	93	0.053	2.247	(3.1769,0.0015)	(-0.6521,0.5833)	127
Oct7	115	0.044	2.361	(3.3144,0.0009)	(-0.4333,0.7543)	129
Oct13	106	0.055	2.285	(3.5214,0.0004)	(-0.3115,0.8646)	178
Oct20	103	0.056	2.275	(3.415,0.0006)	(-0.3578,0.831)	126
Oct27	94	0.049	2.249	(3.2942,0.001)	(-0.4418,0.7646)	152
Nov3	85	0.051	2.221	(3.4936,0.0005)	(-0.4801,0.7446)	109
Nov12	100	0.061	2.262	(3.4943,0.0005)	(-0.3278,0.853)	136
Nov17	106	0.045	2.252	(3.513,0.0004)	(-0.372,0.8171)	139
Nov26	111	0.047	2.324	(3.5791,0.0003)	(-0.2401,0.905)	148
Dec1	89	0.046	2.237	(3.5868,0.0003)	(-0.4061,0.8017)	126
Dec17	274	0.046	2.072	(2.6786,0.0074)	(-0.6133,0.5454)	228
Dec28	105	0.052	2.228	(3.5349,0.0004)	(-0.3391,0.8513)	157
Jan2	97	0.068	2.236	(3.6232,0.0003)	(-0.3892,0.8218)	113
Jan10	238	0.060	2.123	(2.1909,0.0285)	(-0.58,0.55)	148
Jan18	102	0.057	2.309	(2.8405,0.0045)	(-0.344,0.8072)	192
Jan26	107	0.050	2.324	(3.2356,0.0012)	(-0.1421,0.9579)	180
Jan31	230	0.069	2.001	(1.857,0.0633)	(-0.7246,0.4001)	152
Feb8	101	0.060	2.253	(3.0705,0.0021)	(-0.3245,0.8426)	231
Feb15	87	0.058	2.191	(3.651,0.0003)	(-0.4765,0.755)	213
Feb23	112	0.044	2.272	(3.0616,0.0022)	(-0.3062,0.8515)	180
Mar1	103	0.053	2.279	(3.0959,0.002)	(-0.3276,0.8367)	148
Mean	117.964	0.052	2.237	(3.0933,0.0043)	(-0.4050,0.7787)	163.786
SD	46.713	0.008	0.074	(0.4346,0.0127)	(0.1288,0.1234)	38.686

Table 9.3: Optimal power law fits for each Emerald snapshot.

MW				Log likelihood		
Date	xmin	D	α	$x^{-\alpha}$ vs $e^{-\lambda x}$ (R,p)	$x^{-\alpha}$ vs $x^{-\alpha}e^{-\lambda x}$ (R,p)	n tail
Oct23	102	0.033	2.286	(3.1454,0.0017)	(-0.3017,0.8536)	238
Nov3	95	0.039	2.351	(3.2167,0.0013)	(-0.2512,0.885)	177
Nov17	100	0.037	2.385	(3.1308,0.0017)	(-0.2238,0.8991)	129
Nov26	113	0.037	2.347	(3.1295,0.0018)	(-0.2709,0.8673)	197
Dec3	97	0.039	2.308	(2.9975,0.0027)	(-0.4291,0.7447)	126
Dec8	102	0.035	2.374	(3.1125,0.0019)	(-0.297,0.8451)	113
Dec17	101	0.034	2.347	(3.1302,0.0017)	(-0.337,0.8186)	177
Dec29	100	0.036	2.266	(3.0127,0.0026)	(-0.4103,0.7554)	250
Jan2	103	0.039	2.284	(2.9507,0.0032)	(-0.485,0.6861)	334
Jan10	112	0.039	2.351	(2.9712,0.003)	(-0.3616,0.7876)	206
Jan18	142	0.041	2.390	(3,0.0027)	(-0.2263,0.8877)	201
Jan26	110	0.033	2.329	(2.9653,0.003)	(-0.374,0.7806)	157
Jan31	97	0.038	2.308	(3.045,0.0023)	(-0.4321,0.7418)	162
Feb8	95	0.039	2.307	(2.9043,0.0037)	(-0.4015,0.7605)	174
Feb15	111	0.038	2.429	(3.0156,0.0026)	(-0.1404,0.9454)	107
Feb23	125	0.049	2.368	(3.1872,0.0014)	(-0.2147,0.9027)	185
Mar1	104	0.037	2.378	(3.0313,0.0024)	(-0.2576,0.8738)	119
Mean	106.412	0.038	2.342	(3.0556,0.0023)	(-0.3185,0.0703)	179.529
SD	12.057	0.004	0.044	(0.0878,0.0007)	(0.0922,0.0703)	57.755

Table 9.4: Optimal power law fits for each Miller snapshot.

Bibliography

[1] Réka Albert and Albert-László Barabási, *Statistical mechanics of complex networks*, Reviews of Modern Physics **74** (2002), no. 1, 47–97.

[2] Jeff Alstott, Ed Bullmore, and Dietmar Plenz, *powerlaw: A Python Package for Analysis of Heavy-Tailed Distributions*, PLoS ONE **9** (2014), no. 1.

[3] Albert-László Barabási and Réka Albert, *Emergence of Scaling in Random Networks*, Science **286** (1999), no. 5439, 509–512 (en).

[4] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy, *Gephi: An open source software for exploring and manipulating networks*, 2009.

[5] Anthony Bonato, *A Course on the web graph*, Graduate studies in mathematics, vol. 89, American Mathematical Society, 2008 (EN).

[6] J.A. Bondy and U.S.R. Murty, *Graph Theory*, Graduate Texts in Mathematics, no. 244, Springer, 2008 (EN).

[7] Chien-Hsun Chen, Chuen-Tsai Sun, and Jilung Hsieh, *Player Guild Dynamics and Evolution in Massively Multiplayer Online Games*, CyberPsychology & Behavior **11** (2008), no. 3, 293–301.

[8] Hyunwoo Chun, Haewoon Kwak, Young-Ho Eom, Yong-Yeol Ahn, Sue Moon, and Hawoong Jeong, *Comparison of Online Social Relations in Volume vs Interaction: A Case Study of Cyworld*, Proceedings of the 8th ACM SIGCOMM Conference on Internet Measurement (New York, NY, USA), IMC '08, ACM, 2008, pp. 57–70.

[9] A. Clauset, C. Shalizi, and M. Newman, *Power-law Distributions in Empirical Data*, SIAM Review **51** (2009), no. 4, 661–703.

[10] Drew Conway, *Modeling Network Evolution Using Graph Motifs*, arXiv:1105.0902 [physics, stat] (2011).

[11] Jörn Davidsen, Holger Ebel, and Stefan Bornholdt, *Emergence of a Small World from Local Interactions: Modeling Acquaintance Networks*, Physical Review Letters **88** (2002), no. 12.

[12] Nicolas Ducheneaut, Nicholas Yee, Eric Nickell, and Robert J. Moore, *The Life and Death of Online Gaming Communities: A Look at Guilds in World of Warcraft*, Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (New York, NY, USA), CHI '07, ACM, 2007, pp. 839–848.

[13] Steve Hanneke, Wenjie Fu, and Eric P. Xing, *Discrete temporal models of social networks*, Electronic Journal of Statistics **4** (2010), 585–605 (EN). MR MR2660534

[14] Haibo Hu and Xiaofan Wang, *Evolution of a large online social network*, Physics Letters A **373** (2009), no. 12–13, 1105–1110.

[15] Leskovec Jure and Krevl Andrej, *SNAP Datasets: Stanford large network dataset collection*, June 2014.

[16] Charles Kadushin, *Understanding Social Networks,*, 1 ed., vol. 1, Oxford university press, 2012.

[17] Zahra RM Kashani, Hayedeh Ahrabian, Elahe Elahi, Abbas Nowzari-Dalini, Elnaz S. Ansari, Sahar Asadi, Shahin Mohammadi, Falk Schreiber, and Ali Masoudi-Nejad, *Kavosh: a new algorithm for finding network motifs*, BMC Bioinformatics **10** (2009), no. 1, 318 (en).

[18] Haewoon Kwak, Hyunwoo Chun, and Sue Moon, *Fragile Online Relationship: A First Look at Unfollow Dynamics in Twitter*, Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (New York, NY, USA), CHI '11, ACM, 2011, pp. 1091–1100.

[19] David Lazer, Alex (Sandy) Pentland, Lada Adamic, Sinan Aral, Albert Laszlo Barabasi, Devon Brewer, Nicholas Christakis, Noshir Contractor, James Fowler, Myron Gutmann, Tony Jebara, Gary King, Michael Macy, Deb Roy, and Marshall Van Alstyne, *Life in the network: the*

coming age of computational social science, Science (New York, N.Y.) **323** (2009), no. 5915, 721–723.

[20] R. Milo, N. Kashtan, S. Itzkovitz, M. E. J. Newman, and U. Alon, *On the uniform generation of random graphs with prescribed degree sequences*, arXiv:cond-mat/0312028 (2003).

[21] Ron Milo, Shalev Itzkovitz, Nadav Kashtan, Reuven Levitt, Shai Shen-Orr, Inbal Ayzenshtat, Michal Sheffer, and Uri Alon, *Superfamilies of Evolved and Designed Networks*, Science **303** (2004), no. 5663, 1538–1542 (en).

[22] M. E. J. Newman, *Mixing patterns in networks*, Physical Review E **67** (2003), no. 2.

[23] N.D. Poor, *Collaboration via Cooperation and Competition: Small Community Clustering in an MMO*, 2014 47th Hawaii International Conference on System Sciences (HICSS), January 2014, pp. 1695–1704.

[24] Cuihua Shen, Peter Monge, and Dmitri Williams, *Virtual Brokerage and Closure: Network Structure and Social Capital in a Massively Multiplayer Online Game*, Communication Research (2012) (en).

[25] Shai S. Shen-Orr, Ron Milo, Shmoolik Mangan, and Uri Alon, *Network motifs in the transcriptional regulation network of Escherichia coli*, Nature Genetics **31** (2002), no. 1, 64–68 (en).

[26] Sara Nadiv Soffer and Alexei Vázquez, *Network clustering coefficient without degree-correlation biases*, Physical Review. E, Statistical, Nonlinear, and Soft Matter Physics **71** (2005), no. 5 Pt 2 (eng).

[27] Seokshin Son, Ah Reum Kang, Hyun-chul Kim, Taekyoung Kwon, Juyong Park, and Huy Kang Kim, *Analysis of Context Dependence in Social Interaction Networks of a Massively Multiplayer Online Role-Playing Game*, PLoS ONE **7** (2012), no. 4.

[28] Michael Szell and Stefan Thurner, *Measuring social dynamics in a massive multiplayer online game*, Social Networks **32** (2010), no. 4, 313–329.

[29] Riitta Toivonen, Lauri Kovanen, Mikko Kivelä, Jukka-Pekka Onnela, Jari Saramäki, and Kimmo Kaski, *A comparative study of social network models: Network evolution models and nodal attribute models*, Social Networks **31** (2009), no. 4, 240–254.

[30] Johan Ugander, Lars Backstrom, and Jon Kleinberg, *Subgraph Frequencies: Mapping the Empirical and Extremal Geography of Large Graph Collections*, Proceedings of the 22Nd International Conference on World Wide Web (Republic and Canton of Geneva, Switzerland), WWW '13, International World Wide Web Conferences Steering Committee, 2013, pp. 1307–1318.

[31] Brooke Foucault Welles, Anthony Vashevko, Nick Bennett, and Noshir Contractor, *Dynamic Models of Communication in an Online Friendship*

Network, Communication Methods and Measures **8** (2014), no. 4, 223–243.

[32] Sebastian Wernicke and Florian Rasche, *FANMOD: a tool for fast network motif detection*, Bioinformatics **22** (2006), no. 9, 1152–1153 (en).

[33] Xiao-Ke Xu, Jie Zhang, Ping Li, and Michael Small, *Changing motif distributions in complex networks by manipulating rich-club connections*, Physica A: Statistical Mechanics and its Applications **390** (2011), no. 23–24, 4621–4626.